Modeling The Heap: A Practical Approach

by

Mark Marron

B.A. Mathematics, University of California Berkeley, 2001

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2008

Dedication

To those who came before.

“If we knew what we were doing it wouldn’t be research.” – Anonymous

Acknowledgments

I would like to thank my advisor Deepak Kapur for his support in letting me pursue the work in this thesis and for his help in the development of the ideas in this work. His focus on precision in the definitions and constructions helped identify a number of important issues that otherwise would have been missed. I would also like to thank my co-advisor Darko Stefanovic, and committee member Manuel Hermenegildo for the time they spent discussing compiler design and abstract interpretation techniques with me. Without this support the results of my research would be far less likely to be of use in optimizing a program and the analysis would certainly be limited to plodding through micro-benchmarks. Finally, I would like to thank Rupak Majumdar with whom I was able to have a number of very enlightening discussions which led to several of the key techniques developed in this thesis.

Without the support of my committee, their willingness to listen to my ideas, engage in discussions, help in solving many technical problems, and to read numerous drafts of papers this work could not have happened. I am fortunate to have worked with them and deeply appreciate their support.

Finally, I would also like to acknowledge a number of my office mates from who have been there to offer useful comments on problems, help debug code, read drafts of papers, and in general put up with me. Thanks Krister, Stephan and Mario.

Modeling The Heap: A Practical Approach

by

Mark Marron

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2008

Modeling The Heap: A Practical Approach

by

Mark Marron

B.A. Mathematics, University of California Berkeley, 2001

Ph.D., Computer Science, University of New Mexico, 2008

Abstract

The ability to accurately model the evolution of the program heap during the execution of a program is critical for many types of program optimization and verification techniques. Past work on the topic of heap analysis has identified a number of important heap properties (connectivity, shape, heap-based dependence information, and region identification) that are used in many client applications (parallelization, instruction scheduling, memory management) and has explored a range of approaches for computing this information. However, past work has been unable to compute this information with the required level of accuracy and/or has not been computationally efficient enough to make the analysis practical. The inability to precisely and efficiently extract this heap information has limited the utility of many proposed optimization techniques, making the optimization technique impractical or severely limited its effectiveness.

In the present work a general-purpose heap analysis technique is proposed. The major objective is to provide the range of heap information needed by the optimization applications stated above, on real-world programs, in a computationally tractable manner. Keeping tractability in mind, a set of properties/information required by client optimization

applications, e.g., automatic parallelization, instruction scheduling, redundancy elimination transformation, stack allocation, pool allocation, object co-location, or static garbage collection, is proposed. A set of design heuristics are selected to ensure that the analysis is fast and precise in the common case, and to ensure good computational performance by reducing precision in less common situations.

This general approach allows the construction of an analysis that satisfies a number of key requirements for producing a practical heap analysis. The analysis can handle most of the Java 1.4 language including major language features (arrays, virtual calls, interfaces, exceptions) as well as the most commonly used standard Java libraries. No restrictions are placed on the heap structures that are constructed and no restrictions on the recursive data structures built or how these structures are connected/shared. The analysis is able to provide precise information needed for a wide range of optimization applications — aliasing, connectivity, shape, identification of logical data structures, and heap-carried data dependence information. Finally in practice, this information is computed efficiently.

The technique has been implemented and used on a range of small to medium size benchmarks from the JOlden and SPECjvm suites as well as a number of benchmarks that were developed during this work. The analysis technique is evaluated using a number of detailed case studies which focus on the heap structures that the analysis is able to identify in the program and how this information can be used in a variety of optimization techniques (focusing on vectorization/loop scheduling, thread-level parallelization, and memory management). Most of these benchmarks are naturally amenable to thread-level parallelization, thus as a more empirical evaluation of the heap analysis results a detailed evaluation of the analysis is done using the results to drive the parallelization of the benchmark programs, resulting in a substantial speedup over the benchmark suites.

The runtime costs of the analysis are evaluated both in terms of the total analysis time and the general scalability of the analysis. The results of this evaluation indicate that the technique presented in this thesis is indeed a general purpose solution to the heap analysis

problem (at least for small/medium size programs) that can be used to efficiently compute precise and useful information for program optimization over a wide range of real world Java programs.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The ability to identify relationships among data structures has a significant impact on the effectiveness of program optimizations. Techniques from diverse areas such as optimizing memory layouts, garbage collection, extracting thread-level parallelism, enhanced instruction scheduling, and the elimination of redundant memory operations have all been shown to require or benefit substantially from improved information about a range of heap properties.

Early work on *alias* analysis [40, 63, 67], the identification of simple *connectivity* information [9, 18, 31, 38] and the explicit modeling of *memory-carried dependence* [10, 14, 34, 35] have been used to improve scheduling and the effectiveness of classic redundancy elimination operations [10, 39, 45, 64]. Other work focused on using more detailed *connectivity* and *shape* to identify which sections of the heap are disjoint and thus can be modified in parallel, which allows the introduction of thread-level parallelism into single-threaded programs [20, 21, 29].

Work on optimizing memory layouts [11, 42, 52] uses information on how data structures are connected and the notion of identifying objects that are part of the same recursive data structure to improve the locality of memory accesses in the program. Other work on memory management has focused on how to use connectivity and lifetime information to either improve the performance of garbage collection [33] or to enable the static collection of some parts of the heap [12, 19, 27, 65].

While work in these areas examined the importance of identifying and utilizing information about various heap properties, the effectiveness of all these approaches was limited (sometimes severely) by the inability to obtain this information with the desired level of precision. The inability to compute precise heap information using simple variable sharing properties [9, 18, 20, 31, 40, 63] motivated the development of more sophisticated heap modeling approaches that tracked the shape and connectivity properties of the entire heap [3, 24–26, 28, 43, 58–60, 68]. These approaches provided significant increases in the level of accuracy, but they still lack the representational capacity to model important structures, such as arrays that point to the same object in multiple indices or unstructured cyclic regions. They also do not provide support for the standard Java libraries, which are used extensively in any non-trivial program (they either ignore them or attempt to analyze them directly, which is computationally expensive and often imprecise). The computational cost of these approaches makes them infeasible for the analysis and transformation of realistic programs. In many of these approaches analysis times are reported in the 100's of seconds even for small programs (a few thousand lines of code), or only simple micro-benchmarks are analyzed. While some of these techniques have been efficiently applied to larger programs, severe restrictions are placed on the behavior of the program (e.g., the program may only build lists without sharing).

1.2 Properties of Interest

Before we begin on the technical portion of the thesis we first want to look at the set of heap properties that we have chosen as important and that we want to capture in our analysis. Below we outline each of the properties that we would like the analysis to be able to provide information on and references to the literature where this property (or similar information) has been used in an optimization of application. Our study of the needs of optimization applications in Section 1.1 provided a fairly small set of properties that are sufficient to support the majority of optimization applications (and most other applications can be supported by post-processing the results of the analysis). These properties can be grouped into roughly three categories: connectivity (aliasing, reachability, interference and shape), locality (region identification), and store properties (dependency and lifetime).

Aliasing. One of the most basic heap properties to model is the concept of *aliasing* between variables [9, 40, 63, 67]. This property is used in almost every optimization or verification technique that requires heap information. It is used heavily in optimizations that perform simple redundancy elimination or scheduling optimizations [39, 45]. Aliasing answers the question: given variables x and y, *may* there exist a memory location that both x and y refer to?

Connectivity. Two related but more general concepts are *reachability* and *interference*. These properties have significant applications in thread-level parallelization transformations [21, 29, 58]. These techniques utilize information on what parts of the heap must be *disjoint* to determine that a given recursive call sequence or loop cannot have any heap-carried data dependencies. Once this determination is made, the calls or loop iterations can be re-written to run on multiple threads in parallel. The *reachability* predicate we consider is: given two memory locations, m_1, m_2, does there exist a (potentially empty) path of pointers that starts at m_1 and ends at m_2? We can also ask about *interference*: given two

memory locations, m_1, m_2, does there exist a third memory location m_3 which is distinct from m_1, m_2, such that m_3 is reachable from m_1 and m_3 is reachable from m_2?

These properties were a major focus of the work in [18, 20, 38, 60]. An interesting feature of these approaches is the detail that they use to represent the paths. At one end of the spectrum is the binary representation in [20] where the only information kept on the paths is their existence. At the other end of the spectrum is the model for the paths used in [18, 25] where the sequence of fields and the number of times each field is taken is modeled.

Shape. A major motivation for the modeling of *reachability* and *interference* was the use of these properties to determine the *shape* of some section of program memory. As with *reachability* and *interference*, the concept of *shape* of a given section of the heap enables a range of thread-level parallelization transformations. Early work on shape analysis [20] focused on the shape of the section of memory that was reachable from a given variable. Using *reachability* and *interference* information, it is possible to determine if the memory locations reachable from a variable are connected in a *Tree* shape, a *DAG (Directed Acyclic Graph)* shape, or a *Cycle* shape. More recent work on shape analysis has focused on more precise methods for identifying the layout of recursive data structures [24, 26, 43, 59, 60].

Regions. A number of recent analysis techniques [3, 28, 41] have looked at the problem of identifying regions of memory that are part of the same *logical data structure*. In particular [41] demonstrated performance improvements that are possible by pool-allocating objects that make up the same data structure in the same section of memory. In [33] the idea of regions and connectivity was used to do garbage collection on small parts of the heap (which are expected to contain large numbers of dead objects) and it was shown that improved region identification had a substantial impact on the speed of the garbage collector. The approach for identifying which objects are in the same *logical data structure*

is a heuristic notion based on a definition of what constitutes a recursive data structure and when do two sections of the heap contain equivalent data structures. For example, [41] uses allocation sites, call site context sensitivity, and local flow analysis to identify individual data structures, while [28] uses the points-to sets computed in a preprocessing step to partition the heap into sections. Finally [3] uses the separating conjunction ($*$) as an implicit partition of the heap into disjoint sections.

Memory-Carried Dependencies. Often the interest in the shape or connectivity of the heap is to use the information as a proxy for identifying memory-carried data dependencies. That is, if the analysis can determine that two portions of the heap are disjoint then we can be certain that operations performed in one of the portions cannot affect the values in the other portion of the heap. However, in general it would be desirable to directly track which portions of the heap may be modified by any given program statement. This was partially explored in [34] but the imprecision of the heap model and the lack of field sensitivity in the analysis severely limited the utility of the results. More recent work [53] has shown promise but is still highly constrained by the base heap model.

Object Lifetimes. The final property that is of interest to us is the identification of object lifetimes. The ability to determine when an object dies allows a number of improvements in managing a program's memory. The first work on identifying object lifetimes attempted to identify allocations that are local to a single procedure [19, 65] (escape analysis). Other work has generalized this to handle situations where objects may escape the local frame and to identify objects that have similar lifetimes [13, 41, 52]. More recent work has focused on how to statically identify the lifetime of individual objects to allow them to be collected without the use of a garbage collector [12, 27].

1.3 Contributions

The major contribution of this thesis is the heap analysis technique. While there is extensive prior work on modeling the program heap this work represents the first approach that is capable of analyzing small to medium size real-world Java programs (that make use of the range of language features, dynamic methods, arrays, the use of standard libraries, exceptions, etc., present in real Java programs) in an efficient and precise manner. As our experimental results show, we are able to efficiently produce precise heap information over a wide range of programs that build and manipulate a diverse set of heap structures (lists, trees, cyclic structures, and collections from the standard libraries, both with and without sharing). The approach taken in this thesis is to identify and address the myriad problems that arise when attempting to analyze the program heap and to address each of them in turn in whatever manner is the most practical. Thus, throughout this work we draw on a range of prior work and develop a number of novel techniques according to what is needed to solve the problem at hand.

From the previous work we build on the ideas of a *storage shape graph* as presented in [9], we heavily use the concepts of materialization of recursive structures and the application of a focus operation from the Three-Value Logic Analysis (TVLA) work [59, 60], we develop a notion of separation and identification of logically related portions of the heap based on the ideas in [9, 18, 41, 54], and we also draw from the ideas of shape in [20]. We extend these ideas in a number of novel ways:

- In Chapter 3 we extend the classic *storage shape graph model* into a parametric heap analysis framework by allowing it to be augmented with a number of novel instrumentation predicates.
- In Chapter 8 we use these instrumentation predicates to define materialization and focus operations based on the ideas in [59, 60], which allows us to precisely model recursive data structures and sharing in the heap.

- In Chapter 9 we develop a modified set of local and interprocedural data-flow equations to efficiently analyze a program with these extended storage shape graphs.

- In Chapter 11 we use the natural explicit store representation present in the storage shape graph model and we show how heap-carried data dependence information can be precisely and efficiently computed.

In addition to solutions based on the extension of existing ideas with new properties this work also introduces several new techniques for dealing with problems that arise when analyzing the heap. In particular these techniques address a number of problems which have not been previously addressed (in any significant way) in the previous work on heap analysis.

- In Section 3.3 we introduce a novel method for modeling unstructured sharing between heap structures. While most previous work has focused on the precise modeling of recursive data structures, very little attention has been paid to the problem of modeling sharing between two distinct data structures or collection objects.

- In Section 3.4 we further improve the support for modeling sharing by introducing a second concept for sharing that not only allows us to precisely model if two collections or data structures *may* share but also to precisely model on what objects they *must* share.

- In Chapter 6 we explore a novel heuristic for identifying and grouping logically related sections of the heap based on a number of definitions which characterize *uniform* sections of recursive data structures and contents of unbounded structures.

- In Chapter 10 we present a novel technique for modeling the semantics of collection objects in the from the standard collection libraries in Java.

To evaluate the effectiveness of the approach presented in this thesis and to show that it is indeed "a practical technique for modeling the heap" we extensively evaluate how the analysis performs (both in terms of the utility of the information produced and the computational cost of obtaining this information) on a range of small and mid size programs. We draw from the JOlden Suite [37] and the SPECjvm98 suite [62] as well as developing some of our own benchmark programs to ensure that our tests cover a diverse and representative set of heap structures that are commonly built and techniques that are used to traverse or update these structures.

Our case study evaluations of the analysis on the benchmarks (Chapter 5) and the parallelization results and the experimental evaluation of the computational costs (Chapter 13) demonstrate that we have produced a powerful, general-purpose analysis technique that can be used to analyze small and medium size, real-world Java programs. While we have not achieved our final goal of producing a fully general analysis (as the analysis framework is still immature and there are open questions about scalability to large programs), this thesis represents a substantial advance in the state of the art of heap analysis. We have succeeded in developing a heap analysis method that is able to precisely model a wide range of heap properties that are useful for program optimization. We can support nearly the full Java language including many non-trivial features such as full support for the standard Java collection classes. We have produced a (fairly robust) full implementation of this method and verified that it can indeed produce information (in a computationally tractable manner) that is useful in optimization applications. Thus, this work addresses (at least for smaller programs) the long-standing open problem of producing accurate and useful information about the shape and connectivity of the program heap and provides a solid foundation for continued work to scale this solution up to much larger programs.

1.4 Organization

The main body of the thesis is roughly divided into two major parts such that it is accessible to a range of audiences. Chapter 2 through Chapter 5, along with Chapters 6 and 7 as needed, are designed to provide the reader interested in the potential applications of this analysis with an understanding of what information can be extracted by the analysis and how it might be applied to various optimization applications. The second half of the thesis, Chapters 8-11, are devoted to how the properties we are interested in are actually computed. Since our main goal was the development of a practical heap analysis technique, work was done on many aspects of the problem: the model definitions, the abstract semantic operations, and the interprocedural analysis, all of which are discussed in the second part of the thesis.

Chapter 2

Concrete Model and Storage Shape Graph

The approach taken in this work is based on the *abstract interpretation* framework proposed by Cousot in [17]. This work formalized the concepts of data-flow analysis that are used for many classic compiler analysis techniques [45]. The technique of *abstract interpretation* takes an abstract model that captures *interesting* information about possible concrete states of the program (in this work information about the program heap) and then uses this model to statically determine all possible configurations of the concrete program at each point in the program. The text [49] contains an extensive and accessible description of the abstract interpretation framework and the requirements of the models. In Chapter 7 and Chapter 9 we present a variation on this framework that is specialized for the particular aspects of statically analyzing the heap.

Given this general framework we first (in this chapter) define a simple abstract model and define the relationships between the abstract models and concrete program states that they represent. Thus in this chapter we introduce the *Labeled Storage Shape Graph* model which is an extension of the classic *Storage Shape Graph* model described in [9, 38]. The

labeled storage shape graph is a parametric model that can be extended with additional instrumentation properties to improve the precision of the analysis and to allow us to track information that cannot be expressed with a *storage shape graph*. Once we have defined this model (and given a formalization of the concrete program heap) we can construct an *abstraction* function and a *concretization* function that relates sets of concrete program states to elements (models) in the abstract domain of *Labeled Storage Shape Graphs*.

The *labeled storage shape graph* is a natural base representation for a large number of the connectivity and shape properties that we are interested in, and it is simple to add additional instrumentation properties to track other useful information about the concrete heap. The *labeled storage shape graph* works by using the nodes in the graph to represent regions of the concrete heap or variables, and the edges represent sets of pointers or variable targets.

2.1 Concrete Program Model

For the purpose of having a well-defined problem in this work we decided to restrict our analysis to the Java language and associated libraries. However, the technique should be easily generalizable to most memory-safe, object-oriented programming languages. In particular we minimize the Java-specific idioms, and simplify the dataflow equations and abstract semantic operations by preprocessing the Java source program into a simple (fairly generic) intermediate representation.

The analysis is performed on a core sequential imperative language (MIL, Mid-level Intermediate Language) with memory allocation and destructive updates. The MIL language supports a wide range of object-oriented features including classes, interfaces, and dynamic dispatch (on argument types as well as receiver class), and has support for annotating library methods with implementations or specialized semantic functions. That is,

a library code can either be given as a direct implementation or we can specify a special semantic function in the analysis that directly implements the required semantics of the method. See Chapter 10 for an example from the `java.util` library.

The computational core of the MIL language is statically typed and has the standard range of primitive operations, method invocations, conditional constructs (`if`, `switch`), exception handling (`try`, `throw`, `catch`), and the standard looping statements (`for`, `do`, `while`). The state modification and expressions cover the standard range of program operations (load, store, and assign, along with logical, arithmetic, and comparison operators). Below is a subset of the MIL grammar:

$$
\begin{array}{lll}
\text{atom} & ::= & \text{var} \mid \text{literal} \\
\text{expr} & ::= & \text{atom} \mid \text{atom} + \text{atom} \mid \texttt{new}\ \text{type}(\text{atom}, \ldots, \text{atom}) \mid \text{atom}.f \\
 & & \mid \text{atom}.m(\text{atom}, \ldots, \text{atom}) \mid \text{var}\ \texttt{instanceof}\ \text{type} \mid \ldots \\
\text{stmt} & ::= & \text{var}=\text{expr} \mid \text{var}.f=\text{atom} \mid \texttt{break} \mid \ldots \\
\text{contol} & ::= & \texttt{if}(\text{atom})\ \text{block}\ \texttt{else}\ \text{block} \mid \texttt{while}(\text{atom})\ \text{block} \mid \ldots \\
\text{block} & ::= & (\text{stmt} \mid \text{control})*
\end{array}
$$

The semantics of the language is defined using an environment that maps variables into values, and a store that maps addresses into values. We refer to the environment and the store together as the concrete heap. We model the concrete heap as a labeled, directed multi-graph ((*V*)*ariables*, (*O*)*bjects*, (*R*)*eferences*) where each variable $v \in V$ is a variable in the environment, each $o \in O$ is an object in the store, and each labeled directed reference edge $r \in R$ represents a reference (a pointer between objects or a variable reference). Each reference has a storage location label that is either a field identifier from the program or an integer $i \in \mathbb{N}$ (the integers label the pointers stored in the collection or arrays). For a reference $p \in E$ we use the notation $a \xrightarrow{p} b$ to indicate that the object/variable a points to b via the reference p. We also use the notation $a \overset{\psi}{\rightsquigarrow} b$ to denote that there is a non-empty path of references, $\psi = \langle r_1 \ldots r_k \rangle$, in the concrete heap that starts at a and leads to b. We

use the notation $\psi \subseteq_{path} \{r_1, \ldots, r_j\}$ to denote that every reference that appears in the path ψ is in the set of references $\{r_1, \ldots, r_j\}$.

A *region* of memory $\Re$ is a subset of the objects in memory, with all the pointers that connect these objects and all the cross-region references that start or end at an object in this region. Formally, let $C \subseteq O$ be a subset of objects, and let $P = \{p \mid \exists a, b \in C, a \xrightarrow{p} b\}$ and $R_{cross} = \{r \mid \exists a \in C, x \notin C, a \xrightarrow{r} x \vee x \xrightarrow{r} a\}$ be respectively the set of internal and cross-region references for C. Then a region is the tuple (C, P, R_{cross}).

If we treat a subgraph, (C', P'), of a given region, (C, P, R_{cross}) where $C' \subseteq C \wedge P' \subseteq P$, as an undirected graph we can define *weakly-connected* components of the graph in the usual way. For a region $\Re = (C, P, R_{cross})$ and objects $a, b \in C$, we say a and b are *connected* in $\Re$ if they are in the same weakly-connected component of the subgraph (C', P'). Objects a and b are *disjoint* in $\Re$ if they are in different weakly-connected components of the subgraph (C', P').

2.2 Example Concrete Heap and Labeled Storage Shape Graph

Figure 2.1 shows a concrete heap that uses several types, a simple dummy data object, `DN` $\equiv$ {`int val`}, a class that can be used to build a linked list containing `DN` objects, `LN` $\equiv$ {`DN data, LN next`} and a pair class - `PN` $\equiv$ {`LN first, LN second`}.

In Figure 2.1 the variable `l` points to the head of linked list of `LN` objects, the first element in the list has a *null* data entry, the second two elements in the list share their `(d)ata` targets with the array pointed to by variable `A` and the next two elements in the list reference the same `DN` object which is also referred to by the variable `y`. These elements in the linked list are also pointed to by the `(f)irst` and `(s)econd` fields of a `PN` object and this object is referred to by the variables `p1`, `p2`. Finally, the last two

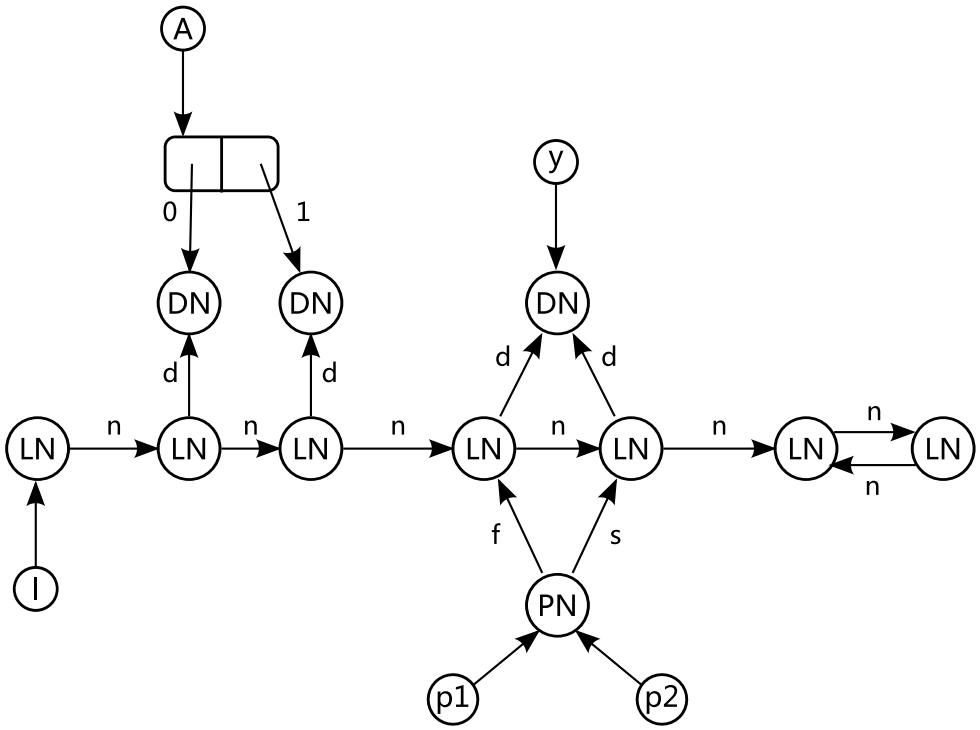


Figure 2.1: Concrete Heap

`LN` objects point to each other creating a cycle at the tail of the list. Thus, this particular heap contains a range of interesting structures and sharing relations including an acyclic list segment, a cyclic list segment, an array, and objects that are aliased as well as objects that are unaliased.

Figure 2.2(a) shows one possible partition for this particular concrete heap into regions. There are many possibilities for this partition but we selected this particular one to illustrate the properties of the model and because it is very similar to what the heuristics in the analysis would produce. The optimal partitioning seeks to group related objects into the same recursive data structures and to group objects that are stored in the same collections into the same regions. We describe in detail how this is done in Chapter 6.

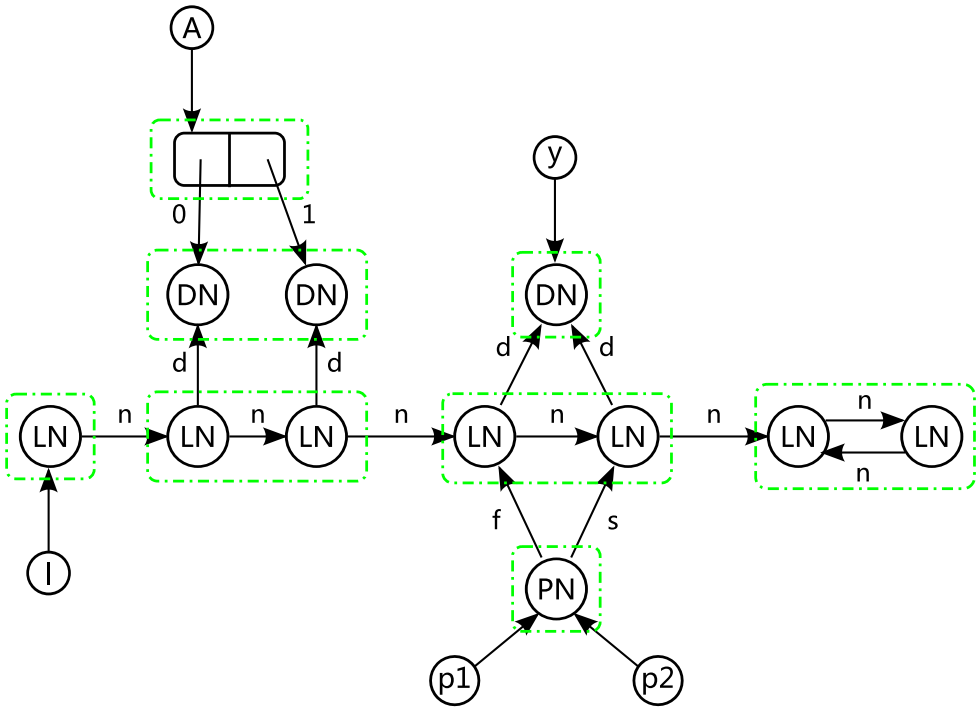
We have created regions for the single `PN` object, the `DN` object pointed to by `y` and the first `LN` object (referred to by `l`). We have also chosen to group the 2^{nd} and 3^{rd} elements

in the list together as a recursive structure as well as grouping the 4^{th} and 5^{th} together. We group the two `DN` objects stored in the array together since they are stored in the same collection. Finally, we have grouped the two cyclic `LN` objects into the same region. These regions are shown in Figure 2.2(a) as dotted boxes (in green if color is available) surrounding the nodes which have been grouped into the same region.

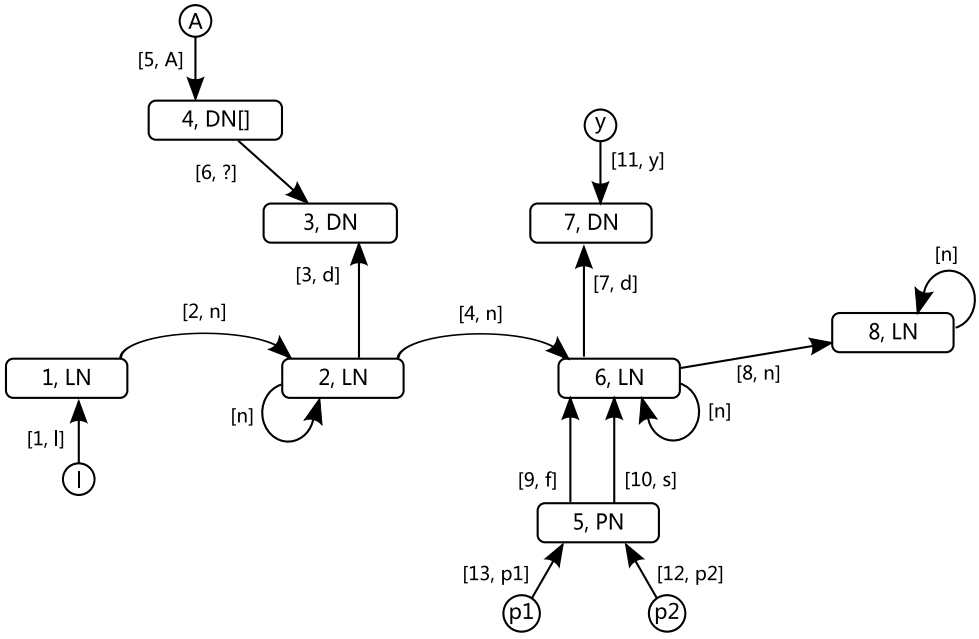
By abstracting the objects in the regions with nodes and the pointers between the regions with edges we get the *Labeled Storage Shape Graph (lssg)* model shown in Figure 2.2(b). Each node and non-self edge is given a unique integer identifier and labeled with the types of objects contained in the region (or a variable name). The edges are also given unique integer identifiers and are labeled with the storage location (*offset*) that the reference is stored in (a field identifier, a variable name, or the special offset `?` for pointers stored in an array). The only information retained in this model is the types of the objects in the regions and some coarse connectivity information provided by the graph structure and the edge storage location offsets.

It is clear that we have lost a considerable amount of information that was present in the concrete heap using this basic abstraction. For example, it is no longer clear that the pointers in array `A` point to different `DN` objects, and the same holds for the `(d)ata` entries in the linked list represented by node 2. Additionally, we have lost relational information between various references in the heap, for example we do not know if the variables `p1` and `p2` alias, if the `(f)irst` and `(s)econd` fields of the pair refer to the same object, different objects, of if there is a path from the object the `(f)irst` field refers to the object that the `(s)econd` field refers to. Finally we know that there is some sort of internal connectivity in the three regions that we identified as containing `LN` objects (based on the self edges) but we do not know if this internal connectivity indicates cyclic structures or only simple acyclic lists.

The loss of this information due to a lack of expressive power in the model has a substantial impact on the utility of this approach to support the optimization applications



(a) Concrete Heap With Regions Marked



(b) Abstraction Heap Graph

Figure 2.2: Concrete Heap With Partition and Basic Abstraction

we are interested in. In Chapter 3 we introduce a number of *instrumentation properties* that will allow us to more precisely express various properties of the concrete heap which are needed to effectively and consistently be able to perform the program optimizations we are interested in.

2.3 Abstraction and Concretization

Given a concrete heap (V, O, R) and a partition, Ξ, of the objects in this heap into regions we can abstract the concrete heap graph into a *labeled storage shape graph* $(\hat{V})$*ariables,* $(\hat{N})$*odes,* $(\hat{E})$*dges,* $\hat{L_n}$, $\hat{L_e}$), where $\hat{L_n}$ is a map from the nodes to their labels (the integer identifiers and sets of type names) and $\hat{L_e}$ is a map from the edges to their labels (the integer identifiers and offsets). This is done using the following abstraction function (α):

Definition 1 (Abstraction (α_{lssg}) given partition Ξ)**.** *Given a concrete heap* (V, O, R) *and* Ξ *a partition of the objects O we can construct a* labeled storage shape graph *that approximates the given heap as follows:*

1. *For each variable v in* V *a variable node is added and labeled with the variable name.*
2. *For each region* $\Re = (C, P, R_{\text{cross}})$ *in* Ξ *we add a node and label it with the types of all the objects in the region.*
3. *For each reference r in* R *where* $a \xrightarrow{r} b$ *we add an edge from the node that represents a to the node that represents the region in* Ξ *containing b. The edge is labeled with the storage location the reference is stored at in a (variable name, field offset or for integer indices the special offset ?).*

To concretize a *labeled storage shape graph* we enumerate all possible concrete heaps along with all possible partitions of the objects in each heap and then check if the heap and

partition are consistent with the abstract *labeled storage shape graph*. If for a given heap there is a partition such that it is consistent (as defined below) then it is in the concretization otherwise we discard it.

Definition 2 (Concretization (γ_{lssg}))**.** *Given the* labeled storage shape graph $g = (\hat{V}, \hat{N}, \hat{E}, \hat{L_n}, \hat{L_e})$*, a concrete heap* $h = (V, O, R)$*, and partition* Ξ *are consistent with g if:*

1. *There exists a 1-1 function* $\Pi_v : V \mapsto \hat{V}$*, a 1-1 function* $\Pi_o : O \mapsto \hat{N}$ *and a function* $\Pi_r : R \mapsto \hat{E}$*.*

2. $\forall o, o' \in O$ *(*o, o' *in the same partition of* Ξ *iff* $\Pi_o(o) = \Pi_o(o')$*).*

3. *The types of the objects in each region of the partition* Ξ *are a subset of the type names in node label under the map,* Π_o $(\{\text{type} \mid \exists$ *object* $o \in \Xi, o$ *instanceof* $\text{type}\})$ $\subseteq \hat{L_n}(\Pi_o(o))$.types*.*

4. *The edge labels are consistent with the storage locations under the map* Π_r *(given a reference r the* storage location *of reference* $r = \hat{L_e}(\Pi_r(r))$.offset*).*

5. *The graph connectivity properties are consistent under the mappings* Π_v, Π_o, Π_r *(*$\forall$ *objects* $o_1, o_2 \in O$ *and pointer* $p \in R$ *s.t.* $o_1 \xrightarrow{p} o_2$ $\exists$ *edge* $e = \Pi_r(p)$ *where e starts at* $\Pi_o(o_1)$ *and ends at* $\Pi_o(o_2)$*). Similarly for the variable references (*$\forall$ *variables* $v \in V$*, objects* $o \in O$ *and references* $r \in R$ *s.t.* $v \xrightarrow{r} o$ $\exists$ *edge* $e = \Pi_r(r)$ *where e starts at* $\Pi_v(v)$ *and ends at* $\Pi_o(o)$*).*

We can view the concretization function (γ_{lssg}) as a filter on the set of all concrete heap graphs that are feasible for a given *labeled storage shape graph*. By making the language used to describe this filter more expressive (by extending the language that is used for the node and edge labels) we can increase the precision of the analysis. The extended label language we present in Chapter 3 and Chapter 4 introduces a number of new labels which imply additional consistency properties of the regions and reference sets that the nodes and

edges in the *labeled storage shape graph* concretize to. These additional restrictions allow a more precise description of the state of the program at each point during its execution.

Chapter 3

Extended Storage Shape Graph

In this chapter we enumerate additional instrumentation properties that we use in the labels of the *labeled storage shape graph* (*lssg*). Using our running example concrete heap, Figure 2.1 in Chapter 2, we demonstrate how these properties can be used to more precisely identify the feasible states of the program.

3.1 Basic Instrumentation Properties

Types. As in the *lssg* approach we track the types of the objects that each node abstracts. This information is particularly important in Java programs, which often heavily use virtual methods. Thus this type information is critical to resolving the targets of the calls, to allow for more accurate analysis results and a more precise call graph for use by later optimizing passes. Each node in the graph represents a region of the heap (which may contain objects of many types). We track these possible object types with a set of type names for each node.

Offsets. Each edge in the graph represents a variable target or a set of pointers stored in some object field or an array. The edges are labeled with the storage location (*offset*) that the reference is stored in (a field identifier, a variable name, or the special offsets described below for pointers stored in arrays or collections). We assume that an edge always represents references that are stored in the same class of storage location. Thus, for instance, the same edge will never abstract one pointer stored at field `f` and another pointer stored at field `g`.

For arrays and collections we use several special names for the locations where pointers are stored. If we have a container (an array or a collection from `java.util`) and the program is not actively indexing in the container (either with an integer index or an `Iterator`) then we use the *offset* `?`. When an array or collection is being actively indexed by the program we partition the elements into three groups of *offsets*. We order the contents based on the natural iteration order (for integer indexing the $\leq$ on $\mathbb{N}$ and for `Iterator` objects the order items are returned from the collection). Using this order we give the single unique pointer stored at the current indexing position the *offset* `at` (at index), all of the pointers that come before the current indexing position the *offset* `bi` (before index), and all of the pointers that come after the current indexing position the *offset* `ai` (after index).

Linearity. In the basic *lssg* model each region is abstracted by a node in the abstract heap graph and no information is provided about the number of objects abstracted by the node. In our example several of the regions we picked contained only a single object (the objects pointed to by `l` and by `y`) while others contained many objects (the regions representing parts of the list and the data elements stored in the array `A`). The same situation occurs with the edges. The edge with the offset `first` (edge 9) represents a single pointer while the edges with the `(d)ata` offsets (edges 3, 7) represent multiple pointers.

This multiplicity information is critical to performing strong updates during the anal-

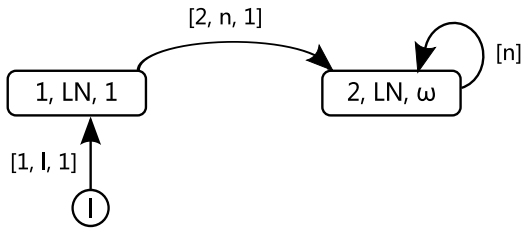


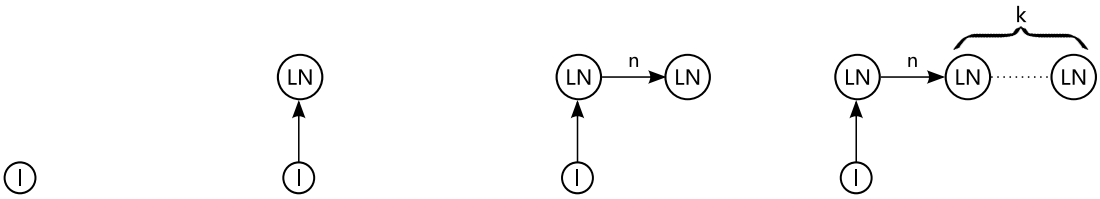
Figure 3.1: Abstract List with Linearity

ysis [9, 59] but can also be useful for some optimization applications (such as stack allocation). To model this information we introduce a property called *linearity* which has two possible values: *1*, which indicates that the node (edge) concretizes to either 0 or 1 objects (pointers) and the value ω, which indicates that the node (edge) concretizes to any number of objects (pointers) in the range $[0, \infty)$.

The inclusion of a multiplicity of 0 as a value for both *linearity* values allows us to compactly represent many possible heap structures. Consider the case of a simple linked list pointed to by the variable `l` where the list may be of any length (including empty). Using the *linearity* definitions we can represent this (and similar situations) with a single abstract graph shown in Figure 3.1.

From the *lssg* in Figure 3.1 we can concretize all possible (acyclic for the purposes of this discussion) concrete linked lists by varying the actual number of concrete objects we place in each region (in a manner consistent with the linearity property). Figure 3.2 shows the various concrete heaps we get as we vary the instantiation of the linearity property. In Figure 3.2(a) we have set the number of objects in both regions to 0 resulting in the empty list, Figure 3.2(b) shows the concrete heap that results when we allow exactly one object in the first region and 0 in the second, Figure 3.2(b) is the concrete heap that results from placing exactly one object in each of the regions and Figure 3.2(d) shows the concrete list that is created as we place k concrete objects in the second region (for $k \in [2, \infty)$).

With the *linearity* property we can update our base *lssg* approximation of the heap



(a) Empty: both 0 (b) Single: 1, 0 (c) Two: both 1 (d) Many: 1, $k \in [2, \infty)$

Figure 3.2: Concretizations of Abstract List with Linearity

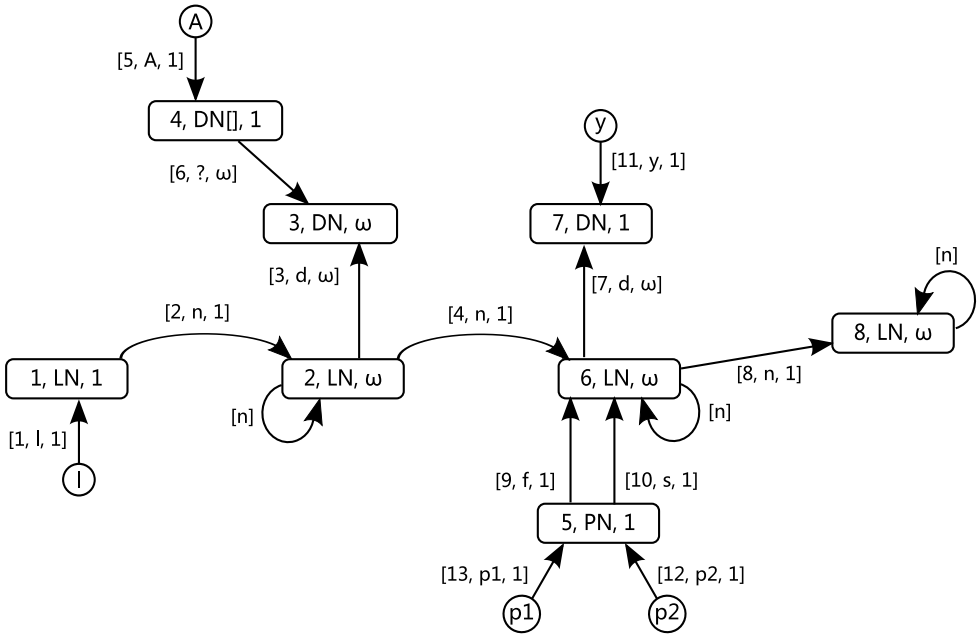


Figure 3.3: Labeled SSG Extended With Linearity

to track the number of objects (pointers) each node (edge) represents. This is shown in Figure 3.3 where we have set the *linearity* of the nodes pointed to by `y`, `l`, `A` and `p1`/`p2` to *1* (indicating that in all concretizations these nodes represent at most one object each). We can also set the *linearity* of the edges with the `(f)irst`, `(s)econd` labels and the `(n)ext` edge from the first `LN` node to *1* (indicating that each of these edges represents at most one reference).

3.2 Abstract Layout

Based on the existence of self edges we can make some simple inferences about the connectivity of the region abstracted by a given node (if there are no self edges then there is no internal connectivity). However, as in our example heap we cannot determine if the nodes with self edges represent acyclic structures or if they represent regions with cycles.

To remove this ambiguity we define a set of structural predicates on the regions of the concrete heap which track commonly used data structure layouts and that we can map to shape labels for the nodes in the *lssg*. Given a region $\Re = (C, P, R_{cross})$ (Section 2.1), we define several structural predicates on the graph (C', P') (where $C' \subseteq C \wedge P' \subseteq P$) to indicate what kinds of traversal patterns a program can use to navigate through the data structures in the region. A region admits a traversal type if there exists a subregion that satisfies the corresponding layout predicate. Note that these structure predicates are *not mutually exclusive*, in particular that *Tree* structure $\Rightarrow$ *List* structure $\Rightarrow$ *Singleton* structure.

Definition 3 (Concrete Region Structure Predicates)**.** *Given a region* $\Re = (C, P, R_{\text{cross}})$ *and assuming* a, b, c *are objects, and* ϕ, ψ *are paths in the concrete heap graph, we define the structure predicates on the region* $\Re$ *as follows:*

- *Cycle Structure if* $\exists a \in C', \phi \subseteq_{\text{path}} P'$ *s.t.* $a \overset{\phi}{\rightsquigarrow} a$.
- *MultiPath Structure if* $\exists a, b \in C', \phi, \psi \subseteq_{\text{path}} P'$ *s.t.* $(a \neq b) \wedge (\phi \neq \psi) \wedge (a \overset{\phi}{\rightsquigarrow} b) \wedge (a \overset{\psi}{\rightsquigarrow} b) \wedge$ *neither* ϕ *nor* ψ *is a prefix of the other.*
- *Tree Structure if* $\exists a, b, c \in C' \wedge p, p' \in P'$ *s.t.* $a \xrightarrow{p} b \wedge a \xrightarrow{p'} c \wedge b \neq c$.
- *List Structure if* $\exists a, b \in C' \wedge p \in P'$ *s.t.* $a \xrightarrow{r} b \wedge a \neq b$.
- *Singleton Structure holds for all regions.*

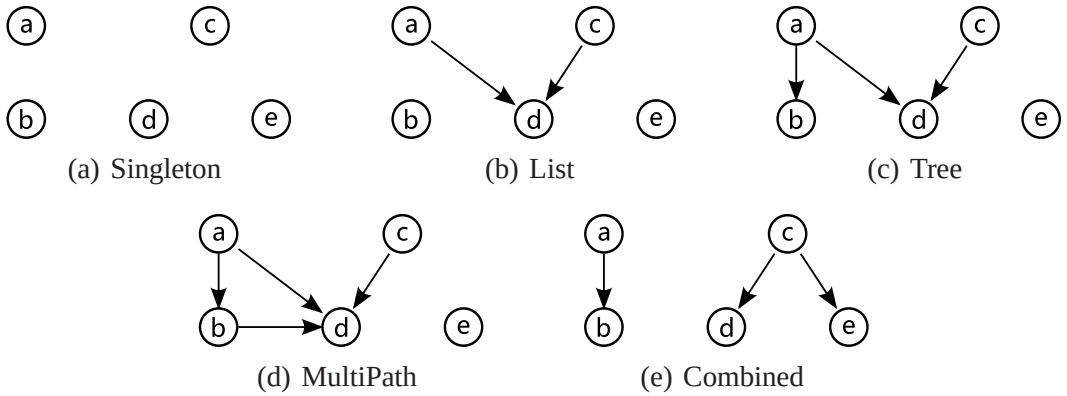


Figure 3.4: Concrete Regions and Structural Predicates

Figure 3.4 shows several concrete regions. Since we are interested in the most general way a program could traverse a region of the concrete heap we must assume that a program variable could begin its traversal of the region at any of the objects in the region. Thus, the figures omit the program variables. Figure 3.4(a) shows a concrete heap with the objects (a,b,c,d,e). Since there are no edges connecting these cells the only way a program can traverse them is by individually referencing each object, thus it only satisfies the *Singleton* structure predicate. Figure 3.4(b) shows a concrete heap that satisfies the *List* structure predicate (both $a \rightarrow d$ and $c \rightarrow d$). It also satisfies the *Singleton* structure predicate since a program can always treat the object as if they were disconnected. Figure 3.4(c) shows a concrete heap that satisfies the *Tree* predicate (a,b,d) as well as the *List* and (trivially) *Singleton* predicates. Figure 3.4(d) adds a pointer, $b \rightarrow d$, that changes the region to satisfy the *MultiPath* predicate (a,b,d) as well. Finally, we note that Figure 3.4(e) satisfies the *Tree* structural predicate. This is due to the existential nature of the structural predicates. Even though the subregion consisting only of a,b does not satisfy the *Tree* predicate the subregion consisting of c,d,e does satisfy the predicate and thus the entire region a,b,c,d,e satisfies the *Tree* structural predicate.

Based on the structural predicates we define a set of instrumentation properties for the nodes in the abstract domain.

Definition 4 (Abstract Layout Properties)**.** *For a node n with a* layout *property label in* {(S)ingleton, (L)ist, (T)ree, (M)ultiPath, (C)ycle} *the concrete* region $\Re$ *is a consistent concretization of n provided:*

- *if n has a* Singleton *Layout, then* $\Re$ only *satisfies the* Singleton *structural predicate but not the* List, Tree, MultiPath, *or* Cycle *predicates.*
- *if n has a* List *Layout, then* $\Re$ *satisfies the* Singleton *or* List *structural predicates but not the* Tree, MultiPath, *or* Cycle *predicates.*
- *if n has a* Tree *Layout, then* $\Re$ *satisfies the* Singleton, List *or* Tree *structural predicates but not the* MultiPath, *or* Cycle *predicates.*
- *if n has a* MultiPath *Layout, then* $\Re$ *satisfies the* Singleton, List, Tree, *or* MultiPath *structural predicates but not the* Cycle *predicate.*
- *if n has an* Cycle *Layout, then any of the structural predicates may hold in* $\Re$.

Using these definitions we can update our abstract domain with the abstract *layout* information (shown as *S, L, T, D, C* labels in the nodes). Figure 3.5 shows the resulting abstract heap graph. In this figure we have labeled the region representing the first element of the list as having a *(S)ingleton* layout since this node abstracts a single object with no self pointers. The node abstracting the contents of the array A also has the *(S)ingleton* layout since the two objects abstracted by this node do not have any pointers between them. The nodes in the abstract graph which, respectively, represent the 2^{nd}, 3^{rd} (node 2) and the 4^{th}, 5^{th} (node 6) LN objects have been given the *(L)ist* abstract *layout* since the regions abstracted by the nodes all have a single, unique successor and there are no cycles. On the other hand the node representing the region made up of the two LN objects at the tail of the list (node 8) is given the *(C)ycle* abstract *layout* since the objects in the region abstracted by the node both have a unique successor but there is a cycle in the region.

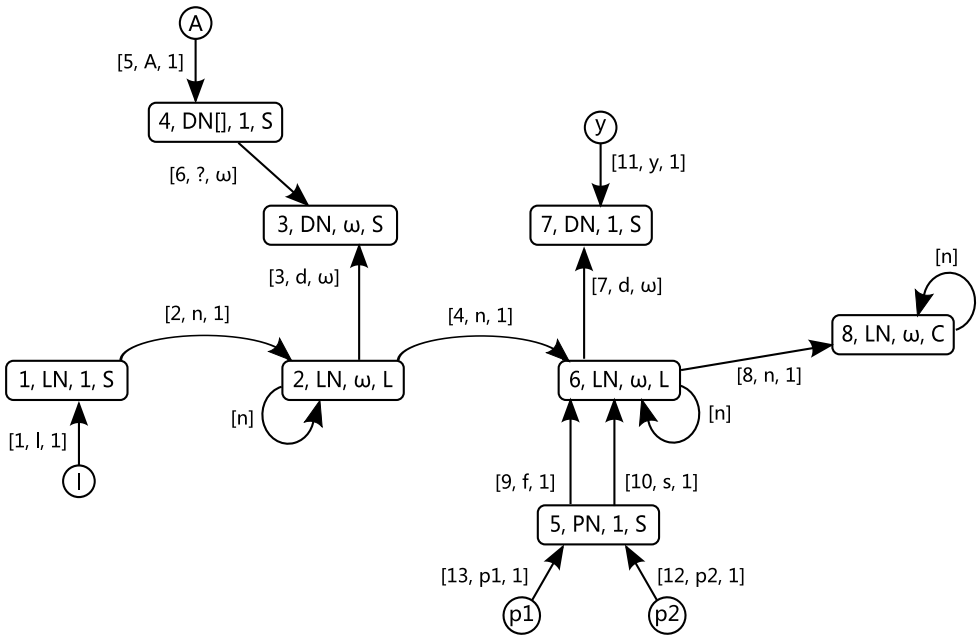


Figure 3.5: Labeled SSG Extended With Abstract Layout

An important observation about the formulation of shape/layout on a per node basis (instead of based on the entire section of the heap reachable from a variable) is that it enables the analysis to be much more precise in the tracking of shape information and prevents the lack of precise shape information in one part of the heap from impacting the ability to precisely represent the shape of other parts of the heap. In our example even though the tail of the list is a cyclic structure the model is still able to represent the fact that the first part of the structure the `LN` objects form is indeed a list and that the `DN` objects stored in this list structure are not connected in any way (since the nodes abstracting them are both *(S)ingleton* and there are no edges between these nodes). See Chapter 5 for more examples where tracking the shape on a per region enables improved precision.

3.3 Connectivity and Interference Properties

In the graph model the edges represent sets of pointers or variable targets and each of these edges ends at a node which represents some region of the heap consisting of (potentially) many different objects and data structures. An important pair of questions is then, do any of these references (pointers/variables) point to the same object in the region and do they potentially point into a connected component in the region? This question can be asked either about references that are abstracted via different edges (e.g., the query, *may* `p1` *alias* `p2`) or about references that are abstracted by the same edge (e.g., the query, *may* any of the pointers stored in array `A` alias).

In our running example we have several of these situations. We know that the contents of array `A` (edge 6) all refer to distinct objects (and further that there is no way that the object referred to by `A[0]` is reachable from `A[1]`), that the `DN` objects stored in the 2^{nd} and 3^{rd} `LN` objects (edge 3) are again distinct but we also know that they point to the same objects as are stored in the array `A`. We also know that the variables `p1` and `p2` alias, that the `(f)irst` and `(s)econd` fields of the `PN` object do not alias but refer to objects in the same list data structure and finally that the `DN` fields of the 4^{th}, 5^{th} `LN` objects contain aliasing pointers (edge 7).

None of this information is captured in the basic shape graph model as there is no means to differentiate edges with aliasing references from edges with non-aliasing references. The edge abstracting the pointers stored in the `DN` fields of the 2^{nd} and 3^{rd} `LN` objects (edge 3) has the same label properties as the edge abstracting the pointers in the `data` fields in the 4^{th}, 5^{th} `LN` objects (edge 7) even though the aliasing properties of these two pointer sets are very different.

To model these properties we introduce two instrumentation properties on edges, one to track reachability relations between references that are abstracted by different edges in the model (*connectivity*) and one to track reachability relations between references that

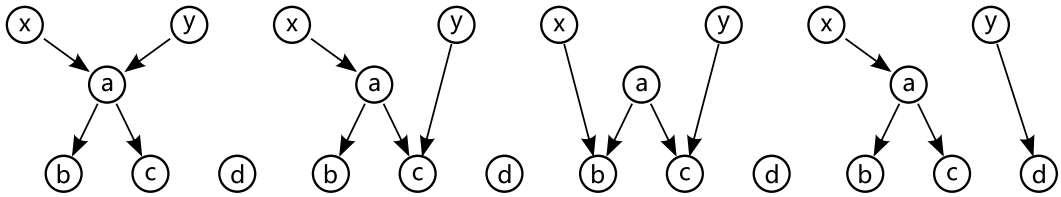
are abstracted by the same edge in the model (*interference*). As with the definitions of the shape properties, we define several predicates on the concrete heap regions and then define the property labels in the abstract domain based on the satisfaction of some set of these predicates.

Definition 5 (Concrete Reference Relation Predicates)**.** *Given a concrete region of the heap* $\Re = (C, P, R_{\text{cross}})$ *and incoming references* $r, r' \in P$ *such that* r, r' *refer to objects* $o_1, o_2 \in C$ *respectively, we define the following* relation *predicates on the references:*

- r, r' alias *with respect to* $\Re$ *if:* $o_1 = o_2$.
- r, r' *are* related *with respect to* $\Re$ *if:* $(o_1 \neq o_2) \wedge (o_1, o_2$ *are in the same* weakly-connected *component of* (C, P)*).*
- r, r' *are* unrelated *with respect to* $\Re$ *if:* o_1, o_2 *are in different* weakly-connected *components of* (C, P).

Figure 3.6 has several examples for the concrete reference relations. Each figure shows a region of the concrete heap, where $C = \{a, b, c, d\}$ and $P = \{a \rightarrow b, a \rightarrow c\}$. In the first example, Figure 3.6(a), we have a concrete heap with two variables x and y that both point to the same concrete object ($R_{cross} = \{x \rightarrow a, y \rightarrow a\}$) thus, according to the definitions above, *alias*. In Figure 3.6(b) the variables x and y point to different objects ($R_{cross} = \{x \rightarrow a, y \rightarrow c\}$) but there is a path from object *a* to object *b*, thus they are in the same *weakly-connected component* of the region, Chapter 2.1, and are *related*. Figure 3.6(c) shows a sample concrete heap where x and y refer to the same *weakly-connected component* ($R_{cross} = \{x \rightarrow b, y \rightarrow c\}$), thus they are *related* according to the above definition even though there is no path between them. Finally, Figure 3.6(d) shows an example of the region where x and y refer to disjoint data structures ($R_{cross} = \{x \rightarrow a, y \rightarrow d\}$) and thus are *unrelated*.

We note that our definitions for references that *alias*, are *related* and are *unrelated* are fairly coarse categories. In particular we might want to further refine the concept of *related*



(a) Aliasing (b) Related (c) Related (d) Unrelated

Figure 3.6: Concrete Reference Relations

into multiple predicates, say *reachable from* and *same structure*, to separate the cases in our second and third examples. In our experience the relatively coarse classification scheme presented above (along with good heuristics for selecting the regions) produces very good results. However, as we gain more experience with the analysis this classification may need to be revisited.

Connectivity. Given the definitions for *related* in the concrete heap we can define a series of *connectivity* properties, $conn = \{share, connected, disjoint\}$, on the edges in an abstract graph. The *connectivity* property is then a relation on $\hat{E} \times \hat{E} \times \hat{N} \times conn$.

Definition 6 (Abstract Connectivity Property)**.** *For a node n and edges e, e' the concrete region $\Re$ and the sets of references $R = \{ r \mid \Pi_r(r) = e \}$, $R' = \{ r' \mid \Pi_r(r') = e' \}$ (where Π_r is part of the concretization relation given in Section 2.3) are consistent with n, e, e' if:*

- *if e, e' are* disjoint *then $\nexists r \in R, r' \in R'$ s.t. (r, r'* alias *with respect to $\Re$) $\vee$ (r, r' are* related *with respect to $\Re$).*
- *if e, e' are* connected *then $\nexists r \in R, r' \in R'$ s.t. r, r'* alias *with respect to $\Re$.*
- *if e, e'* share *then any of the* related *predicates holds for the references r, r' where $r \in R, r' \in R'$.*

To represent the connectivity relation in our *labeled storage shape graph* we extend

the representation tuple $(\hat{V}, \hat{N}, \hat{E}, \hat{L_n}, \hat{L_e})$ with a relation *connR* which contains the *connectivity* relation. This results in an extended domain of labeled graph tuples of the form $(\hat{V}, \hat{N}, \hat{E}, \hat{L_n}, \hat{L_e}, connR)$.

To concisely represent the *connectivity* property in the figures each edge label is extended with a list of the identifiers of the other edges in the graph that it has a non-trivial *connectivity* relation with. For each edge label in this list, if the identifier is prefixed with a "!" then the edges are related according to the *share* abstract predicate; if there is no prefix then they are related by the *connected* predicate; all edges whose identifiers do not appear in the list are related according to the *disjoint* predicate.

Figure 3.7 shows our running example abstract heap with the addition of the *connectivity* information. This figure shows how the introduction of *connectivity* information allows the precise modeling of a number of interesting features of the heap. If we look at the edge representing the contents of the array `A` (edge 6) and the edge representing the pointers stored in the `DN` fields of the 2^{nd} and 3^{rd} `LN` objects (edge 3), the model now shows that the pointers represented by these two edges *may* point to the same objects (the !3 and !6 entries in the connectivity lists for the edges). A more interesting situation is between the node (node 6) that represents the 4^{th}, 5^{th} `LN` objects where there are several incoming edges representing the pointers stored in the `(f)irst` (edge 9) and `(s)econd` (edge 10) fields of the `PN` object and the edge representing the `(n)ext` field of a `LN` object (edge 4). The *connectivity* relation between edges 9, 10 shows that although they point at the same node the analysis can now determine that they *must* not alias (the lack of the "!" in the entry for 10/9 in the connectivity lists) but that they *may* point into the same data structure. If we look at the relation between edge 4 and edges 9, 10 we see that the analysis is able to represent that edges 4, 9 *may* represent pointers that *alias* (the "!" on the entries 9/4) while correctly determining that the pointers represented by edges 4, 10 *may* point into the same data structure but that they *must* not *alias* (the lack of a "!" on the entries 10/4 in the connectivity lists).

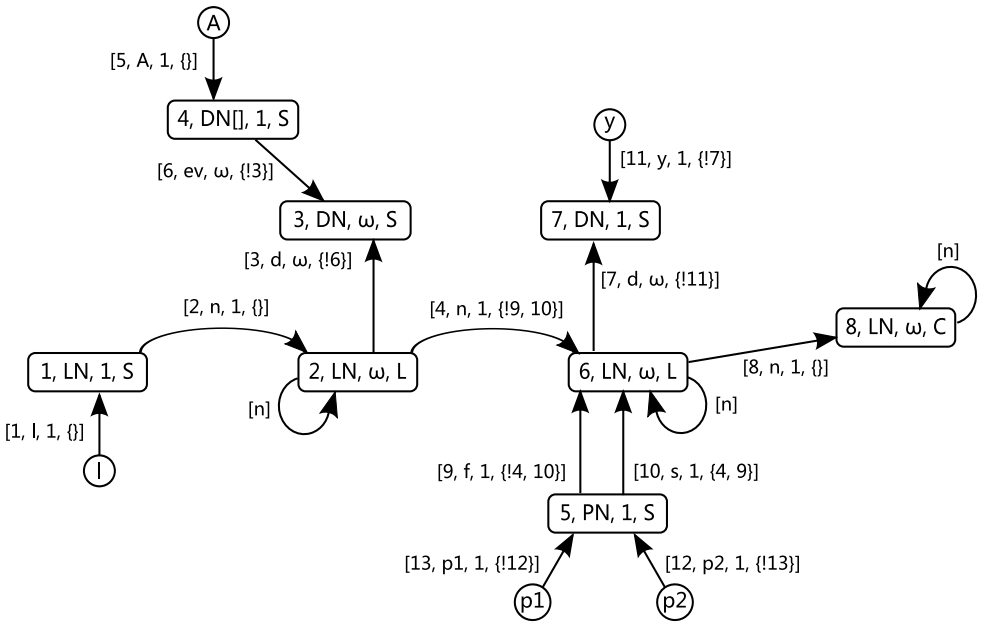


Figure 3.7: Labeled SSG Extended With Abstract Connectivity

Interference. The *interfere* property is closely related to the concept of *connectivity*. While the *connectivity* property tracks *relation* predicates between references that are abstracted by different graph edges, the *interfere* property tracks *relation* predicates between references that are abstracted by the same graph edge. Given the definitions for *related* in the concrete regions, we can define a series of *interference* properties, *interfere* $= \{$*aliasing pointers (ap), interfering pointers (ip), non-interfering pointers (np)*$\}$, on the edges.

Definition 7 (Abstract Interference Property)**.** *For a node n and edge e, the concrete region* $\Re$ *and the set of references* $R = \{ r \mid \Pi_r(r) = e \}$ *(where* Π_r *is part of the concretization relation given in Section 2.3) are consistent with* n, e *if:*

- *if e has the* non-interfering pointers (np) *property then* $\nexists r, r' \in R, r \neq r'$ *s.t. (*r, r' alias *with respect to* $\Re$*)* $\vee$ *(*r, r' *are* related *with respect to* $\Re$*).*
- *if e has the* interfering pointers (ip) *property then* $\nexists r, r' \in R, r \neq r'$ *s.t.* r, r' alias *with*

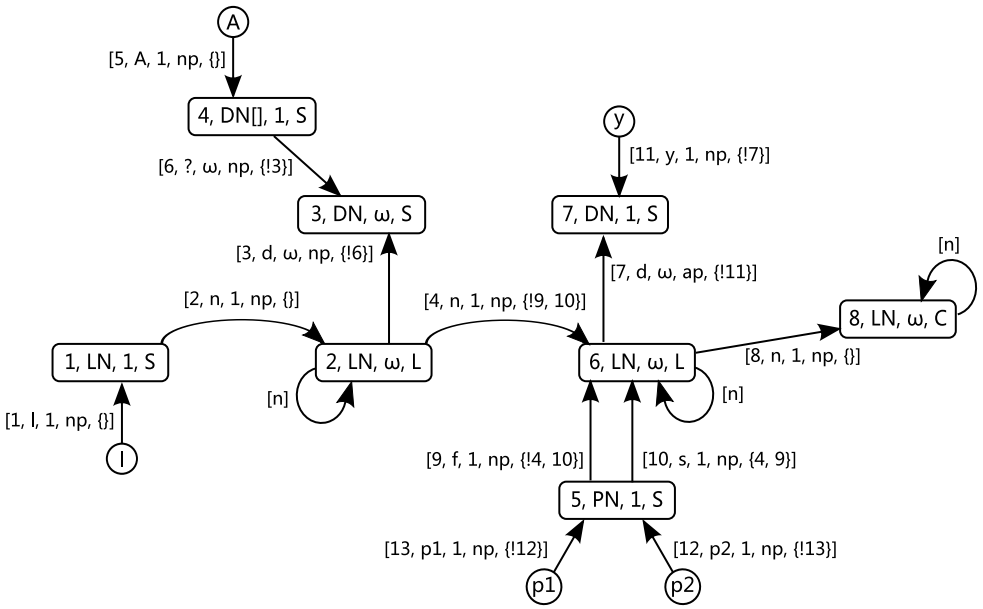


Figure 3.8: Labeled SSG Extended With Abstract Interference

respect to $\mathfrak{R}$.

- *if e has the* aliasing pointers (ap) *property then any of the* relation *predicates holds for the references* $r, r' \in R$.

To represent the *interfere* property in the figures, each edge label is extended with one of the *interfere* predicates $\{ap, ip, np\}$. Figure 3.8 shows the running abstract heap extended with the *interfere* information. Of particular interest in this figure is how the addition of the *interfere* information allows us to determine that the pointers stored in array `A` (abstracted by edge 6) all point to distinct `DN` objects (the *np* label) while the pointers stored in the `data` fields of the 4^{th}, 5^{th} `LN` objects (abstracted by edge 7) *may* point to the same objects (the *ap* label).

3.4 Dominance

The final property related to connectivity/reachablility in the heap graph that we present is *dominance*. This property subsumes the well-known *must-alias* relation but extends the concept from single references to sets of references. In our running example abstract heap we know that the variables `p1` and `p2` *may* refer to the same object (they are related with the *share* abstract property, the *(!13)* and *(!12)* entries in the edge labels) but in the concrete heap that they abstract we know they *must* refer to the same object. While the use of a *must-alias* predicate would be sufficient to handle this situation, if we look at the relation between edges 6 (the contents of array `A`) and 3 (the `data` entries of the 2^{nd} and 3^{rd} list elements) we can see that the *must-alias* predicate is inadequate to precisely express this sharing. In particular we know that the pointers abstracted by edge 3 *must-alias* with the pointers abstracted by edge 6 but also that for every object that is referred to by a pointer in the array `A` there exists a pointer stored in the list that also refers to the object. That is, not only *must* aliasing occur but also *every* reference is aliased.

To track this type of *must* sharing between sets of references we introduce a binary predicate on sets of references in the concrete heap, *edge dominance*.

Definition 8 (Reference Target Sets)**.** *Given a set of incoming references $R = \{r_1, \ldots, r_k\}$ to a region $\Re = (C, P, R_{\text{cross}})$, where $R \subseteq R_{\text{cross}}$, we can define the targets of these reference sets:*

$T = \{\textit{object } o \in C \mid \exists r \in R, r \textit{ points-to } o\}$

Definition 9 (Reference Target Set)**.** *Given reference sets R, R' s.t. $R \cap R' = \emptyset$ with target sets T, T', respectively, we define three predicates based on the relations between T, T':*

- R, R' *are* dominance equal *if* $T = T'$.
- R dominates R' *if* $T' \subseteq T$.

- R, R' *are* dominance disjoint *if* $T \cap T' = \emptyset$.

If we were to translate these concrete predicates into abstract properties in the abstract domain in the natural way we would end up with a binary relation over the powerset of edges in the graph ($\wp(\hat{E})$), which is computationally expensive to process. Thus we introduce a simpler pair of properties to use in the abstract domain. The first is a binary relation on the edges, $domEQ \subseteq \hat{E} \times \hat{E}$ which tracks pairs of edges that represent sets of pointers with the same target sets. The second is a binary relation on the nodes and the edges $nodeDom \subseteq \hat{N} \times \hat{E}$ which tracks for each node which edges represent sets of pointers whose target set is the same as the set of objects abstracted by the node.

Given the definitions for *dominance* in the concrete regions, we define the *domEQ* relation over the abstract edges:

Definition 10 (Abstract Edge Dominance Equality)**.** *For a node* n *and edges* e, e'*, the concrete region* $\mathfrak{R} = (C, P, R_{\text{cross}})$ *and the sets of references* $R = \{r \mid \Pi_r(r) = e\}$*,* $R' = \{r' \mid \Pi_r(r') = e'\}$ *are consistent with* n, e, e' *if:*

- *if* e domEQ e' *then* R, R' *must be* dominance equal.
- *otherwise any of the relations (*dominance equal, dominates, *or* dominance disjoint*) may hold between* R, R'.

Similarly, we use the definitions for *dominance* in the concrete regions to define the *nodeDom* relation.

Definition 11 (Abstract Node Dominance)**.** *For a node* n *and edge* e*, the concrete region* $\mathfrak{R} = (C, P, R_{\text{cross}})$ *and the set of references* $R = \{r \mid \Pi_r(r) = e\}$ *(with target set* T*) are consistent with* n, e *if:*

- *if* e nodeDom n *then* $T = C$.

- *otherwise any of the relations may hold between T and the objects in $\Re$.*

We extend the labels on our graph model tuples ($\hat{V}$, $\hat{N}$, $\hat{E}$, $\hat{L_n}$, $\hat{L_e}$, *connR*) to represent the dominance information by extending the node labels in $\hat{L_n}$ with a list of edges that *nodeDom* the node and by adding the *domEQ* relation to the tuple. The resulting domain with the dominance properties is the set of labeled graph tuples of the form, ($\hat{V}$, $\hat{N}$, $\hat{E}$, $\hat{L_n}$, $\hat{L_e}$, *connR, domEQ*), and we extend the abstraction and concretization operations in the natural way to respect the dominance labels and *domEQ* relation.

In our running abstract heap example we can see several situations where the *dominance* information can provide interesting *must-alias* information as well as information on the subset relations between the contents of data structures on the heap. Figure 3.9 shows how the introduction of *dominance* properties allows us to update the heap with the fact that edges 12, 13 are *domEQ* (marked with a $\sim$ prefix in the connectivity list), which indicates that variables `p1` and `p2` *must* alias. For each node we add a list of all the incident edges that are *nodeDom* with the node. In our example we know that every object in the region abstracted by node 3 has a pointer that refers to it stored in the array `A`. A more interesting pair of facts are the conditional existence relations we have for the variables `A` and `y`. The *nodeDom* property implies that if the variable `y` is null, node 7 represents an empty region of the heap. Similarly the *nodeDom* entry in node 4 indicates that if the variable `A` is null, the node represents the empty region (which is redundant in this case since there is only one edge pointing to node 4).

3.5 Final Labeled Storage Shape Graph

With these properties we are able to precisely describe the information that we need to support the optimization applications we are interested in (Section 1.2 and 1.1). To support the precise modeling of collections and some critical numeric properties, later chapters

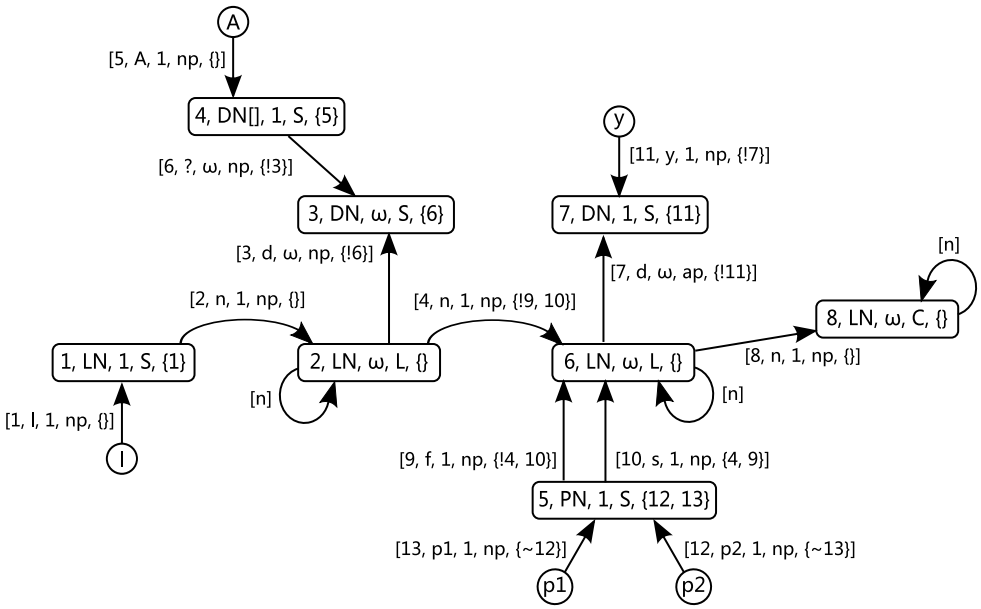


Figure 3.9: Labeled SSG Extended With Abstract Dominance

introduce a few specialized properties for collection nodes and a simple numeric domain. However, the properties introduced in this chapter make up the core of the analysis.

The model introduced here begins with the well known *storage shape graph* from [9, 38] which we then augment with a number of additional labels on the nodes and edges to capture more precise information about the state of the concrete heap than can be represented via the structural information in the graph. The resulting abstract model or *labeled storage shape graph* (*lssg*) is composed of the base graph structure, a set of node labels, a set of edge labels, a connectivity relation, and an edge dominance relation.

The node labels are records of the form [*id types linearity shape domset*] where *id* is a unique identifier, *types* is the set of types for the objects that *may* be abstracted by the node, *linearity* indicates the number of objects that *may* be abstracted by the node, *shape* indicates the possible internal connectivity of the region abstracted by the node, and *domset* lists all the edges that *must* dominate the region abstracted by the node. In our figures, the nodes are labeled with these records.

The edge labels are records of the form [*id offset linearity interfere*] where *id* is a unique identifier, *offset* is the storage location of the references abstracted by the edge, *linearity* indicates the number of references that *may* be abstracted by the edge, and *interfere* indicates the possible *relation* configurations for the references abstracted by the edge. To compactly represent the figures, we include the *connR* and *domEQ* information in the edge labels using a final *conn* list field in the records. This list contains the *id* of any other edge that the given edge has a non-trivial *relation* with (it is *connected* or *share* with it). If the two edges *share*, the id is prefixed with a "!" and if the two edges are *domEQ* the id is prefixed with a "$\sim$".

3.6 Concretization Examples

Throughout this chapter we have been using a sample concrete heap, Figure 2.1 in Section 2.2, to motivate the introduction of a number of label properties that allow us to more precisely model various aspects of this heap in the abstract domain. The result of this construction is the final labeled storage shape graph model described above. In this section we look at several other feasible and one infeasible concretization of the abstract heap in Figure 3.9 in order to gain additional intuition into what the different instrumentation properties we introduced actually imply about the concrete heap.

Empty Heap. Figure 3.10 shows the degenerate empty concretization of the abstract heap. The empty concretization is valid for all heap models since all of the properties we track are either *may* properties (e.g. `l` *may* be null), the *linearity* property always allows the empty concretization ($1 \rightarrow [0,1]$ and $\omega \rightarrow [0,\infty)$) or are universally quantified *must* (the dominance properties, where the empty set always satisfies the universal quantification). Thus, this particular concrete heap is a feasible concretization of our abstract heap.

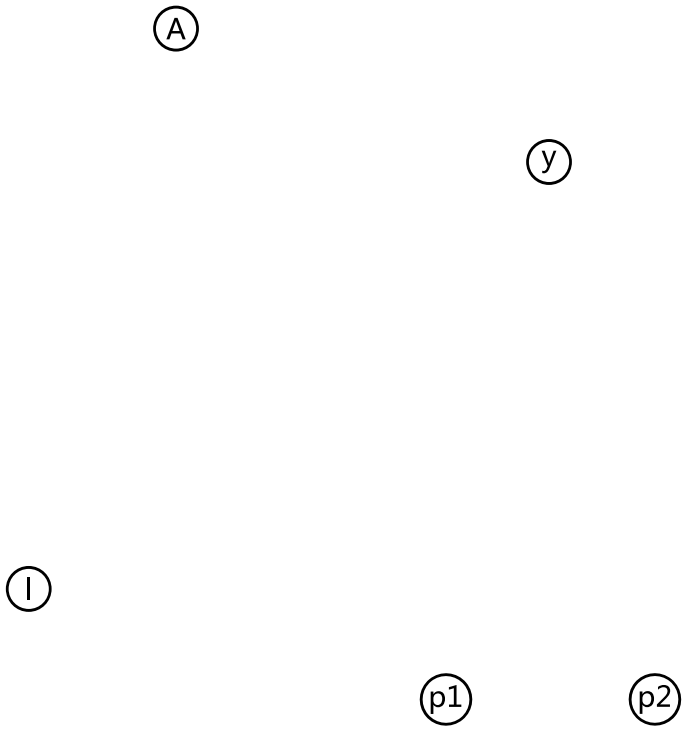


Figure 3.10: Empty Heap

Alternative Heap. Figure 3.11 shows another feasible concretization of our abstract heap graph. In this concretization we note that the tail of the list is acyclic (which is admissible since the *(C)ycle* property allows any concrete layouts) and there is no edge connecting the 4^{th} and 5^{th} `LN` objects (which again is admissible since the *(L)ist* property allows singleton structures). The connectivity of the heap is also altered since pointers `f` and `s` are no longer *connected* but again the abstract *connected* property is a *may* property so a less connected structure is admissible. If we look at the `(d)ata` pointers from the 4^{th} and 5^{th} `LN` objects we notice that one is now null. This is fine even though the edge representing them in the abstract graph has an *interfere* property of *ap* and a *linearity* of ω, because the ω property allows concretizations with a single pointer and the *ap* property allows sets of pointers that are *non-interfering* as well as sets that *alias*.

Thus, this example provides a sense of how the abstract model conceptually provides

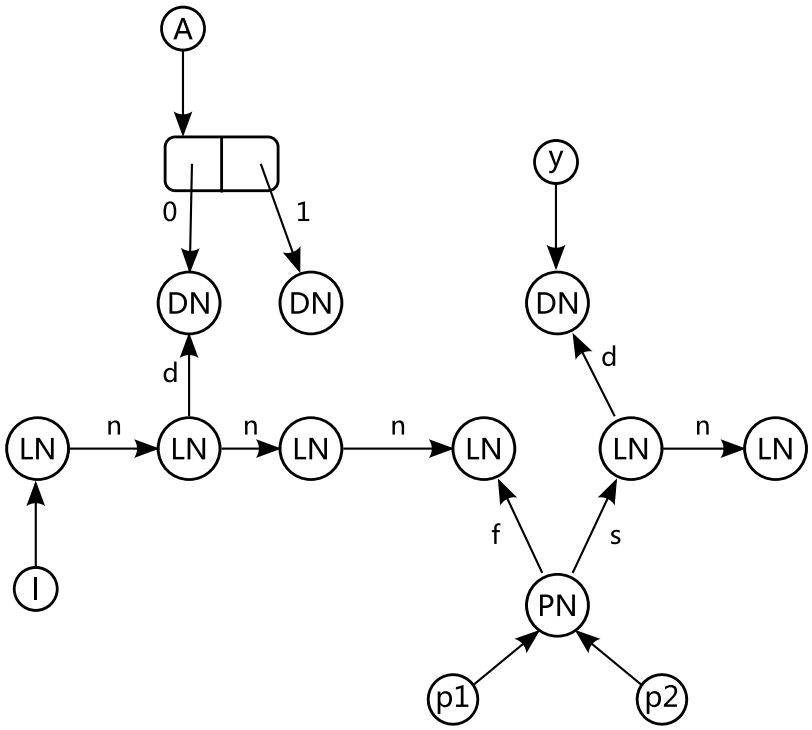


Figure 3.11: Alternative Feasible Heap

upper limits on the connectivity of the concrete heap but in general allows concrete heaps with less connectivity.

Infeasible Heap. Figure 3.12 shows an example of a concrete heap that is not a feasible concretization of our abstract model. This model violates a number of the restrictions present in the abstract model. The most basic is that the pointer stored at `A[1]` does not correspond to any edge in the abstract model. Looking at the pointers stored in the 2^{nd} and 3^{rd} `LN` objects we see they alias which is not consistent with the *np* property of the edge 3 in the abstract model. We also see that the cyclic structure between the 4^{th} and 5^{th} `LM` objects violates the restrictions from the *(L)ist layout* of node 6. Finally, we see that the variable `p2` is null and `p1` is non-null, which is inconsistent with the restriction in the abstract model that edge 12, 13 are *dominance equivalent* which means that `p1` and `p2`

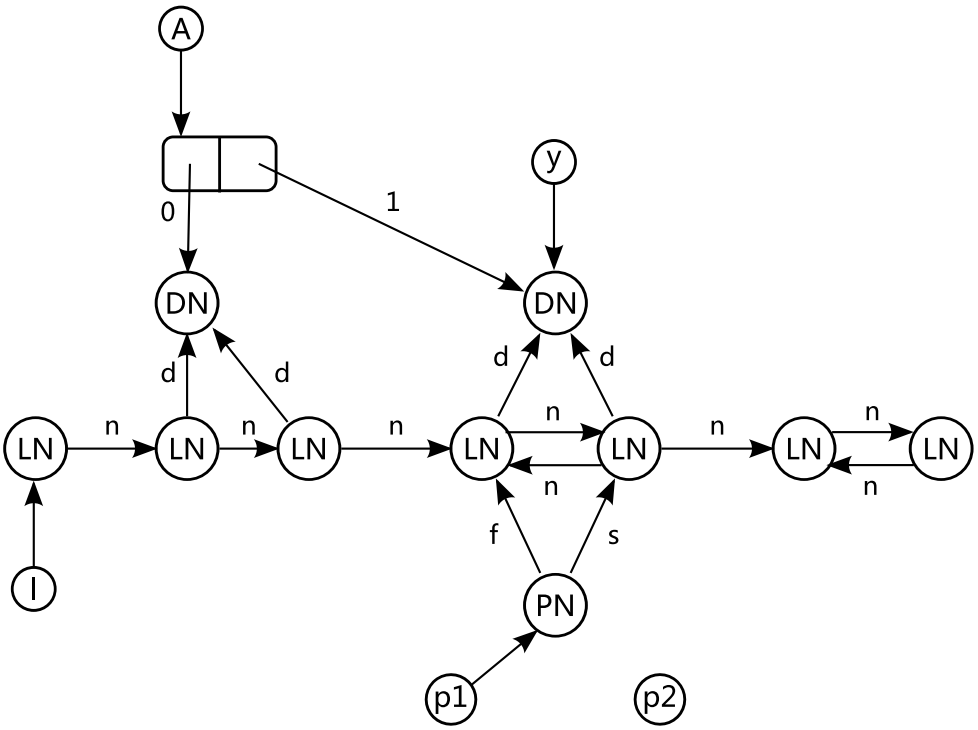


Figure 3.12: Infeasible Heap

must point to the same object (or both be null).

Chapter 4

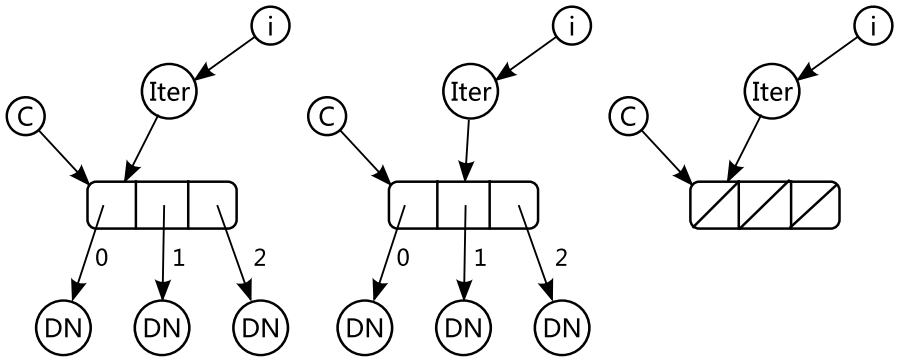
Specialized Extensions and Scalar Domain

This chapter introduces a number of specialized instrumentation labels that are useful when modeling collections (as in `java.util`) and a scalar domain that will allow us to precisely model a number of important types of conditional tests and indexing statements.

4.1 Iteration and Collection Properties

The introduction of the Java collections into the analysis either requires us to analyze the code that is used to implement the collections (which is computationally expensive, can be imprecise, and the constant re-analysis is redundant) or to define higher-level semantics to model the behavior of collections (and arrays). In this section we introduce a number of properties designed to model the semantics of collections and are also potentially useful for optimization applications.

In this section we will use three simple concrete heaps to illustrate the various abstract



(a) Iterator Initialized (b) Iterator Advanced (c) Empty Collection

Figure 4.1: Concrete Collection Heaps

properties that are introduced. All three of our concrete heaps in Figure 4.1 have a single `Vector` object from `java.util` (which we show as a simple sequence of cells each containing a reference), which is non-empty in the first two figures and empty in the third; in each case we have an `Iterator` object (also from the package `java.util`) pointing into the collection.

Active Iterator/Index Variable. The scheme we present for classifying the pointers in a collection is a specific case of the *partitioning functions* that are used in [22] to partition arrays of scalars. The definition we use is only precise when there is a single iterator that is active in a collection. In the case of multiple iterators simultaneously indexing through a collection, our partition must conservatively assume that any relation could hold between the positions of the iterators. The use of more flexible *partitioning functions* would allow our analysis to partition the pointers in a collection even when multiple iterators are being used to index through the collection. However, the use of more general *partition functions* substantially complicates the analysis and we expect that most of the time only a single iterator will be active in a collection. Based on this assumption we opted for the fixed partition.

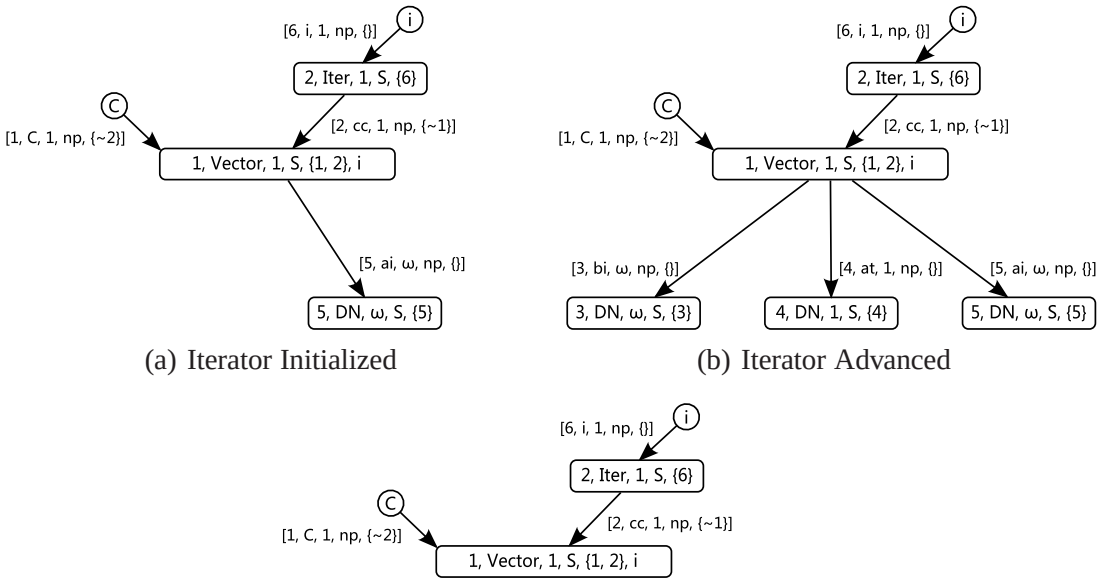
As described in the paragraph on *Offsets* in Section 3.1 we introduce four special edge

labels for edges that abstract the pointers stored in collections and arrays (*?*, *bi*, *at*, *ai*). The *?* field represents a set of arbitrary pointers in the collection while the other offsets are used when an iterator or integer indexing variable is being used to access the collection. The offset *at* is given to the edge that represents the single pointer stored in the location currently being accessed by the iterator/index variable. The offsets *bi*/*ai* represent the pointers that come before/after the current position of the iterator or index variable in the iteration order specified by the collection.

Given our fixed partition approach we need to track which iterator or index variable the partition is relative to. To accomplish this we add to each node that represents a collection a label *activeIndexer*, corresponding to the name of the variable that the current partition of collection entries is based on.

Figure 4.2 shows the abstraction of our three concrete heaps. In the first heap, Figure 4.2(a), the iterator was just initialized so it currently refers to before the first element in the collection and there are no elements that come before it in the iteration order. The set of pointers that come later in the collection are abstracted by the edge with offset `ai`. However, note that the absence of an edge with the label `bi` does not always imply that the iterator is at the beginning of the collection (a feasible concretization of this abstract heap is a vector with all *null* entries before the current iterator position). The second abstract heap is similar to the first except that it does have an edge with the label `bi` which does imply that in any concretization there is at least one entry in the collection before the current iterator position. Our final Figure 4.2(c), is the result of abstracting the empty collection. However, note that the lack of any outgoing edges does not imply that in all concretizations the collection must be empty. A valid concretization is a `Vector` containing only *null* entries.

Empty, At-First. To address the issues that arise with the position of the iterators and the emptiness/non-emptiness of the collections, we introduce two additional abstract pred-



(a) Iterator Initialized (b) Iterator Advanced

(c) Empty Collection

Figure 4.2: Abstract Collection Heaps

icates for the abstract nodes that represent collection objects. These predicates allow us to precisely model the emptiness of a collection and if an iterator object refers to the first element of a collection. These two properties are useful in improving the precision of the analysis as well as allowing some program transformations (such as loop inversion [45]).

The first abstract property is the *empty* enumeration. We treat this predicate as a three-valued relation with values, *true* (shown as e in the node), *false* (shown as $\sim$e in the node) or *unknown* (shown as ?e in the node) and define it as follows.

Definition 12 (Abstract Empty Property)**.** *Given a node n, a possible concretization* $\Re = (C, P, R_{\text{cross}})$ *of n is consistent if:*

- *if the* empty *property is* true *then*

C is a singleton set $\{o\}$*, o is an* `instanceof java.util.Collection` *and o.empty() is* `true`*.*

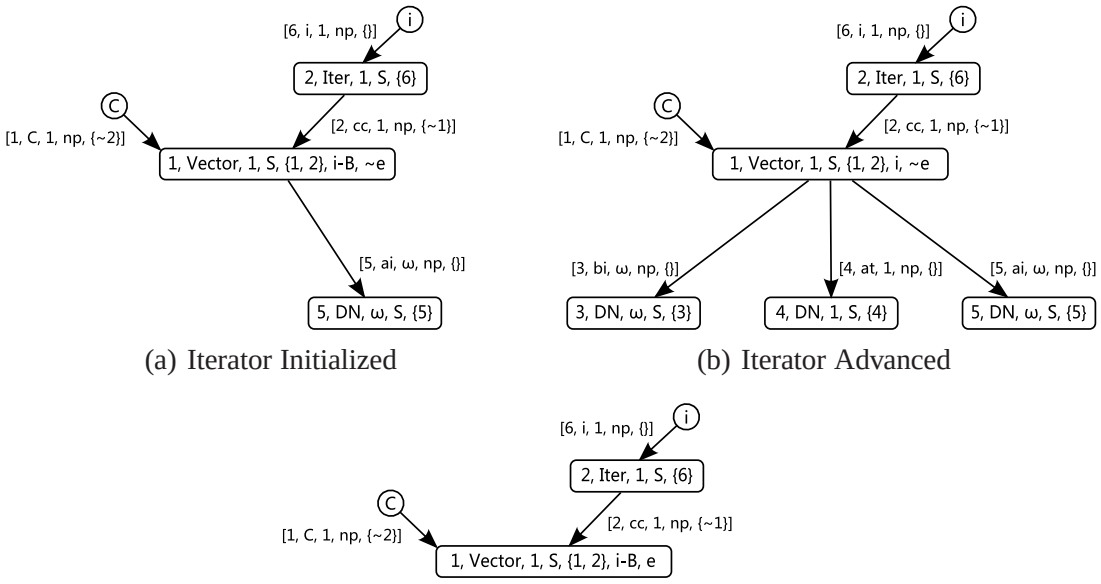
- *if the* empty *property is* false *then*
  C is a singleton set $\{o\}$*, o is an* `instanceof java.util.Collection` *and o.empty() is* `false`*.*
- *if the* empty *property is* unknown *then there are no additional restrictions on* $\mathfrak{R}$*.*

The second property is the *atFirst* enumeration. We treat this property as a two-valued relation with the values, *true* (the active iterator entry is postfixed with `-B`) or *false* (no postfix) and define it as follows.

Definition 13 (Abstract atFirst Property)**.** *Given a node n and an iterator variable I that refers to n a possible concretization* $\mathfrak{R} = (C, P, R_{\text{cross}})$ *of n is consistent if:*

- *if I is the active iterator for n and the* atFirst *predicate is* true *then C is a singleton set* $\{o\}$*, o is an* `instanceof java.util.Collection` *and either o.empty() is* `true` *or I.next() returns the first element in the iteration order of o.*
- *otherwise there are no additional restrictions on* $\mathfrak{R}$*.*

With these additional properties we can update our abstract heap models to more precisely capture the properties of our example concrete heaps. Figure 4.3 shows the extended abstract heaps. The first Figure 4.3(a) shows how the *atFirst* property allows the analysis to determine that the iterator has just been initialized (the *atFirst* property is true, the `-B` postfix of the activeIndexer) and that the call to the *next* method will always succeed (the *empty* predicate is *false*, the $\sim$`e` in the node). The next Figure 4.3(b) shows how the model is able to determine that again the collection is not empty (the $\sim$`e` in the node) and that the iterator is at some position in the collection. The final Figure 4.3(c) shows how the analysis has precisely captured the emptiness of the collection object by marking the node as *must* empty (the, `e`, entry).



(a) Iterator Initialized (b) Iterator Advanced

(c) Empty Collection

Figure 4.3: Abstract Collection Heaps with Empty, First

4.2 Scalar Domain

This section introduces a basic scalar domain that we use to track the results of various boolean expressions, loop guards and simple equality tests. Our main goal for this domain is to be able to accurately model commonly occurring test conditions and the relative positions of index variables in arrays and collections in simple loops while minimizing computational overhead of the analysis.

Representation. Since the properties we are interested in with respect to the scalar values are relatively modest properties and to ensure that the overhead for modeling them is minimal, we use a simple abstraction to model the equality relations between the

integer/boolean local variables, the integer/boolean static fields and the special expression *card*(v) (which tracks the cardinality of containers like arrays and collections in the `java.util` libraries). The relations between these values (variables, constants, and the cardinalities of containers) is modeled using the partition of the set (and the special values 0, 1, *true*, and *false*) based on the *must equal* relation.

Given this general outline for the construction and the set of value symbols $S = \mathit{LocalScalarVars} \cup \mathit{StaticScalarVars} \cup \{\mathit{card}(v) \mid v \in \mathit{LocalVars}\} \cup \{0, 1, \mathit{true}, \mathit{false}\}$ the formal domain for this abstraction is $D_u = \{\pi \mid \pi \text{ is a partition of } S\}$ (the set of all equivalence relations on S). This abstraction allows us to easily track which boolean values must be *true*/*false* so that we can determine if a given abstract value satisfies a boolean guard condition (by checking if the boolean variable is in the same equivalence class as the special symbols *true*/*false*). We can also precisely determine when two index values are equal and if an index value is equal to 0 or the length of an array (or collection) again by simply checking which equivalence classes the variables belong to (and if they are in the same class as the special 0 value).

To support the later uses of this domain we define several simple operations that can be used to simulate the abstract semantics of the boolean and integer values of interest. The most important is the *forgetScalarValue* operation which given a variable erases it from any components of the value domain that it appears in (equivalence relations and the *card* values). We also have the function *addEquality*, which is used to assert an equality relation between two values, combining the equivalence classes that they are in. Finally, we have a *mustEqual* operation which returns *true* if two values *must* be equal (they appear in the same equivalence class).

Abstract Domain Operations With the above definition for the abstract scalar domain we can define the order $\leq_u$ and join $\sqcup_u$ operations as follows.

Definition 14 (Scalar Domain Order)**.** *Given $u, u' \in D_u$, the order operation is:*

$u \leq_u u' \Leftrightarrow \forall v, v' \in S$, *if* v, v' *are in the same partition in* u' *then* v, v' *are in the same partition in u (that is, if an equality* must *hold in* u' *then it also* must *hold in u).*

Definition 15 (Scalar Domain Join)**.** *Given* $u, u' \in D_u$*, the join operation is:* $u \sqcup_u u' = u \cap u'$ *(where* $\cap$ *is the intersection of equivalence relations).*

Abstract Semantics. The semantics of the abstract operations are defined in the expected way. For an assignment of the form `x = y` we first forget all the current information about *x* using the *forgetScalarValue* operation and then use the *addEquality* operation to assert that now *x* and *y must* be equal. More interesting is the order test `b = x < y`. In most situations the domain cannot interpret the precise result of this comparison (since it does not have a generic representation for order, only equality) and so we must conservatively clear all information for `b` and cannot infer any new information. However, in an important class of cases we have information that can allow us to be more precise. In the case where `x`, `y`, `0` are all in the same equivalence class (this often occurs as one of the cases when first entering a `for` loop, e.g., if `x` is the index variable and `y` is equal to the size of the collection which is known to be empty), we know that this test must fail and `b` = `false`.

Chapter 5

Case Studies

In this chapter we examine in detail the analysis results for a number of interesting programs and potential optimizations that can be performed using this information. Due to the space constraints imposed on the size of the figures by a printed medium we focus on programs from our benchmark suite that produce smaller heap structures. While many of the benchmarks we have studied in detail are not impacted by this restriction we refer the interested reader to our online appendix [48] which has figures showing the heap analysis results for a number of programs that are too large to discuss here or that while the analysis produces useful information for optimization the application of this information is similar to other case studies already presented.

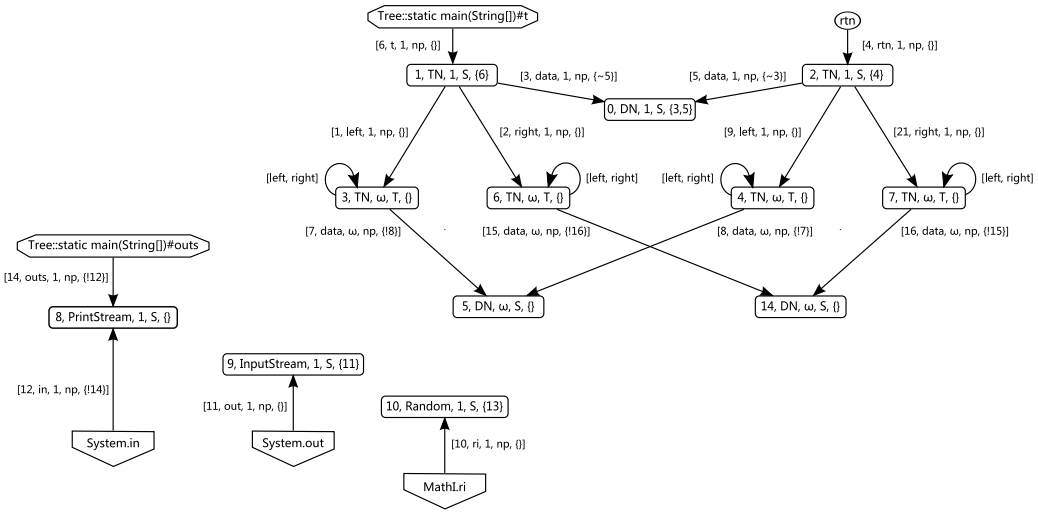
5.1 Intro Tree Example and Model Representation

Our first example comes from a set of simple tree manipulation micro-benchmarks. We use this example to explain several conventions used to simplify the figures, to explain how the figures presented in this paper and the output of the analysis are related, and finally to discuss a number of interesting properties of the analysis.

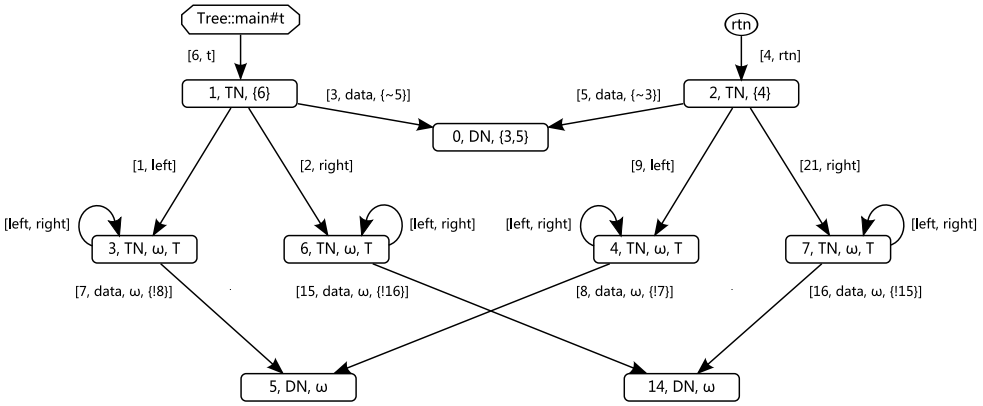
Original Figure. Given a binary tree the `copyTreeShallow` method returns a copy of the tree structure that shares the data elements with the argument tree. Figure 5.1(a) shows the state of the abstract heap at the exit of the of the call `copyTreeShallow`. In this figure we have the variable `rtn` referring to a `TreeNode` object which is the root of the copied tree that is returned. The reference with the label `Tree::static main(string[])#t` is a special label that is introduced to track the reference from the variable `t` in the caller scope. This variable points to the object that is at the root of the argument tree that was passed into the copy routine. We also show a number of static variables `System.in`, `System.out` and a reference to a random number generator `MathI.ri`.

Figure 5.1(a) also illustrates a number of interesting properties that the analysis is able to identify about the heap in this particular method. Some of the properties that the analysis captured in this example are the fact that the argument tree and the result tree have entirely disjoint tree structures although they share the data elements in the tree. In particular the variable `rtn` points to a `TreeNode` object which has `left` and `right` subtree fields, each points into a region of the heap that represents many objects and has a *(T)ree* layout (on the `left` and `right` fields). A more interesting piece of information is the identification of slightly more precise information on the nature of the sharing of the data objects. The abstract heap state shown in the figure shows that while the argument tree and the result tree share some data objects the sharing is only on a subtree basis, that is the `left` subtree of the result may only share data objects with the `left` subtree of the argument tree and similarly the `right` subtree of the result only shares with the `right` subtree of the argument.

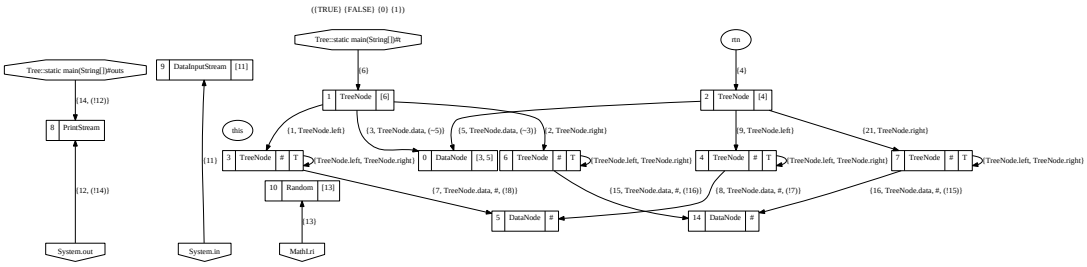
Simplified Figure. In order to simplify the figures we select default values for several of the instrumentation properties and modify the representation of the node/edge labels to omit the components that take on the default values. In the *linearity* domain we pick the



(a) Tree Copy (Full Record)



(b) Tree Copy (Reduced Record)



(c) Tree Copy (Raw Output)

Figure 5.1: Tree Copy

default value to be *1*, the default property for the *layouts* is *(S)ingleton*, for *connectivity* the empty set, for *interference* the *non-interfering* property (*np*) and for *nodeDom* the emptyset. We also omit the *activeIterator*, *emptyCollection* and *atFirst* properties from objects that do not abstract collections (or when the *atFirst* predicate is *false*). We note that the variable reference edges always have the variable name as the *offset* so we omit these as well. Finally, when it is irrelevant to the example at hand we will omit or rename various extraneous references, caller scope references, static variables, etc.

Figure 5.1(b) shows the result of applying these simplifications to the abstract heap graph shown in Figure 5.1(a). If we look at the simplified version of the node that variable `rtn` refers to we see that the explicit denotations for the *linearity* of *1* and the *(S)ingleton layout* have been removed and are now assumed as implicit while the type information (which is always present) and the *nodeDom* set (which is not the default value) are still present in the label. Similarly for the `right` edge (21) in which we have omitted the default values for the *linearity* (*1*), and the *interference* (*non-interfering, np*).

Raw Display Figure. The final Figure 5.1(c) shows the same heap as output by the analysis tool itself. The analysis uses the `dot` program from the `graphviz` suite to automatically generate pictorial graph representations of the heap at *key* points in the program and for output from the interactive analysis debugger. This output format differs slightly from the simplified format that we use for the discussions in this paper in order to ensure that the figures are complete, unambiguous representation of the abstract state. In particular the field identifiers are extended with the class scope that they are declared in, all static/caller scope variables are shown, all local variables (including null valued variables and depending on the program location dead variables) are displayed and as needed the package scope is appended to the type names.

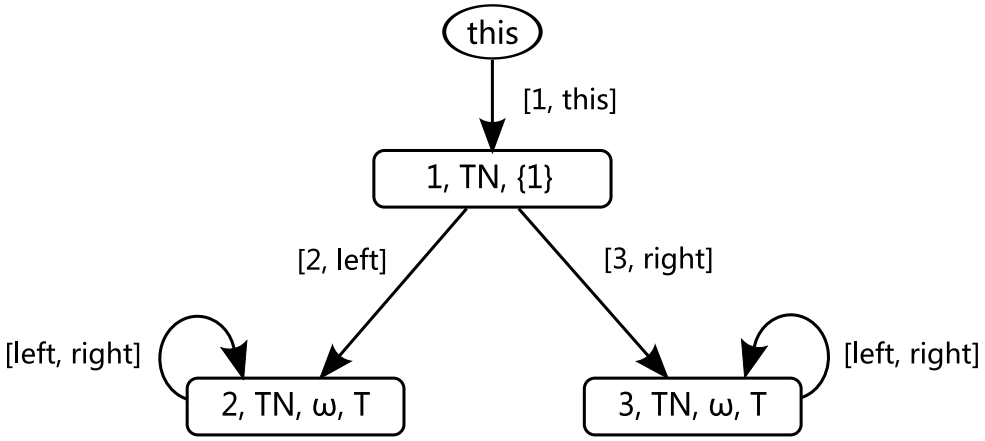
5.2 Select Benchmarks

In this section we look at a few of the more interesting JOlden benchmarks from the standpoint of using the analysis results to drive the optimization of these benchmarks in a number of ways (parallelization, static garbage collection, object collocation) that are generally impractical without the detailed heap information provided by our heap analysis technique. Of course the more detailed information provided by the heap analysis in this work will also improve the effectiveness of traditional optimization techniques (scheduling, redundancy elimination, etc.).

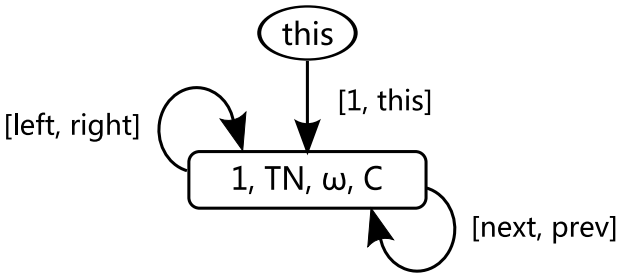
5.2.1 TSP

The first benchmark we consider is `tsp`. This benchmark computes an approximation of the optimal *traveling salesman* path on a tree of points. This is done using a divide and conquer approach where a *tsp* path is computed for each sub-tree and then the paths are merged and the location specified by the current root node is added to the path. The method uses links (via a field `next`) within the nodes to track which node is the next point to visit in the traversal.

Thus the heap at the entry to this method is always the tree structure shown in Figure 5.2(a) while the heap at the exit of the method is the cyclic structure that results from linking the nodes on the path, shown in Figure 5.2(b). This example highlights one of the major strengths of the approach presented in this work. At the entry to the method the state of the heap has a well-defined tree structure which the analysis is able to precisely represent and which allows us to thread-level parallelize the recursive calls to the `left` and `right` subtrees giving a speedup of 3.16 on our quad-core test machine. Conversely at the exit of the method the state of the heap is very irregular and has many non-trivial relations (there is some cyclic path though the objects in addition to the tree structure on



(a) Enter TSP Rec. Call



(b) Exit TSP Rec. Call

Figure 5.2: TSP Recursive Call

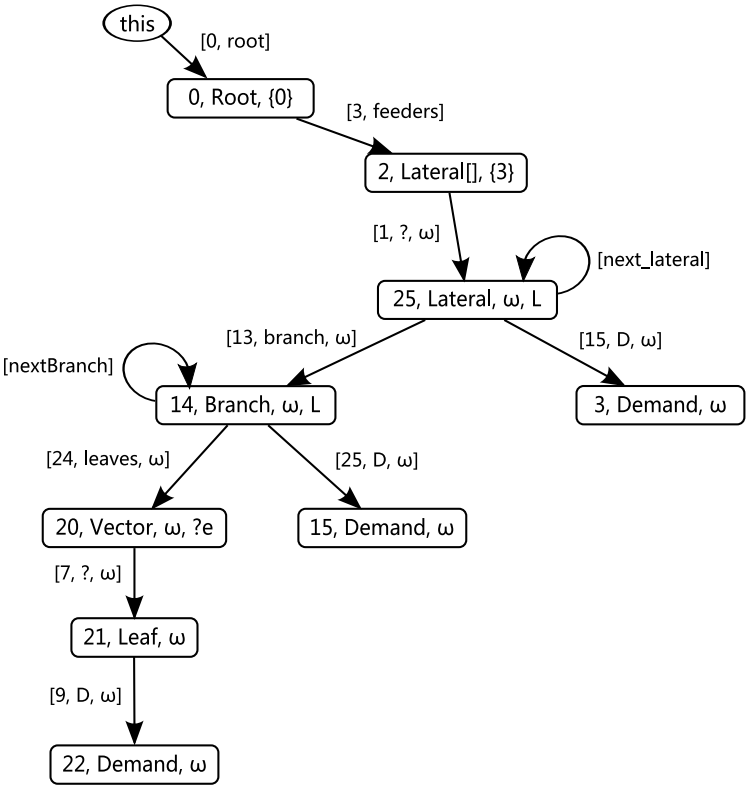
the `left` and `right` fields) and the analysis has shifted to a less precise but much more efficient representation. A related and important feature is how the graph-based model allows the analysis to perform this transition on a region by region basis (the `bh` case study 5.2.5). This allows the analysis to conservatively approximate select regions of the heap for efficiency while maintaining precise information about the state of the program in other regions of the heap where possible.

5.2.2 Power

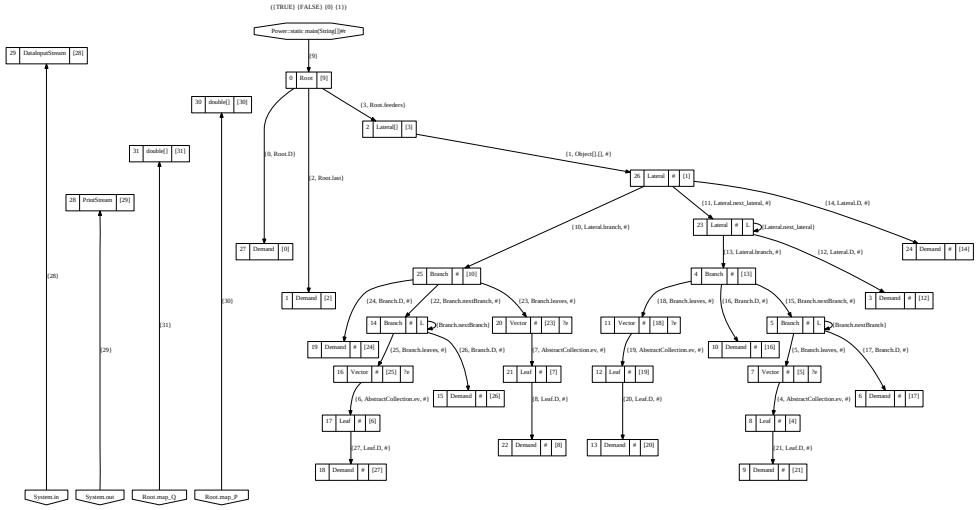
The power benchmark computes the electrical demand of a power grid which is laid out in a tree pattern. The actual implementation is an array of lists of lists of vectors (none of which share in any way), thus this program builds and traverses a similar in structure to the one built by the firewire driver in [3, 68]. The `Root` object is the base of the grid tree and has an array of `Lateral` line objects. Each of these lateral line objects is the head of a list of `Lateral` objects each of which has a pointer to the head of a list of `Branch` objects and each of these branch objects has a `Vector` of `Leaf` objects.

The heap analysis normally represents the head of each list as a separate node from the tail of the list structure but to compact the figure we have compressed the head of the list in with the tail, Figure 5.3(a). We also show the raw output from the analysis with the uncompressed list heads, Figure 5.3(b), which gives a better feel for the shape but due to the larger number of nodes is harder to read.

In Figure 5.3(a) we see that the analysis has identified the `Root` object (node 0) and the initial array of `Lateral` objects (node 2). The edge (edge 1) from the array of `Lateral` objects to the node representing the actual lists of `Lateral` objects has the *interfere* property *non-interfering* (the default value) indicating that each of the pointers abstracted by this edge points to a unique object in a different data structure in the region abstracted by node 25. This means that each of the pointers refers to the head of a unique list. Each of the `Lateral` objects in these lists has a pointer to a `Demand` object and `Branch` object. These pointers are represented by edges 15 and 13 respectively. Again since these edges have the *non-interfering* property we know that they refer to unique `Demand` objects and disjoint lists of `Branch` objects as well. The lists of `Branch` objects, represented by node 14, contain pointers to `Demand` objects (again all uniquely referenced) and `Vector` objects (also uniquely referenced). Finally, we see that each `Vector` object contains references to unique `Leaf` objects and each of these contains a reference to a unique



(a) Power (Compressed List)



(b) Power (Raw Output)

Figure 5.3: Power

`Demand` object.

Thus, the analysis has determined that the heap is an array of lists of lists of vectors with no sharing between any of the lists/vectors as desired. This is sufficient to introduce thread-level parallelism to the processing of the `Lateral` and `Branch` lists which results in a speedup of 3.25 on a quad-core machine.

5.2.3 Em3d

The `em3d` program computes electro-magnetic field values in a 3-dimensional space by constructing a list of `ENode` objects, each representing an electric field value and a second list of `ENode` objects, which represent magnetic field values. To compute how the electric/magnetic field value for a given `ENode` object is updated at each step the `computeNewValue` method uses an array of `ENode` objects from the opposite field and performs a convolution of these field values and a scaling vector, updating the current field value with the result. This example demonstrates the importance of precisely resolving the heap structure so that we can determine that the set of heap locations where the `value` field is written is distinct from the locations that are read, as shown in Figure 5.4. In particular we note that even though the overall heap structure is cyclic the analysis has precisely resolved the bipartite structure and the structures of objects that make up the electric and magnetic field structures.

Figure 5.4 shows the heap structure computed for the `computeNewValue` method. Variable `g` points to a single object of type `BiGrph`, which is the data structure that encapsulates all the objects of interest. The `BiGrph` object has 2 fields, the `hNodes` field pointing to a linked list of `ENode` objects that make up the magnetic field and, the `eNodes` field pointing to a linked list of `ENode` objects that make up the electric field.

Looking at the structure of the heap graph in Figure 5.4 it is apparent that the set of locations read from and written to in `computeNewValue` (Figure 5.5) are disjoint and

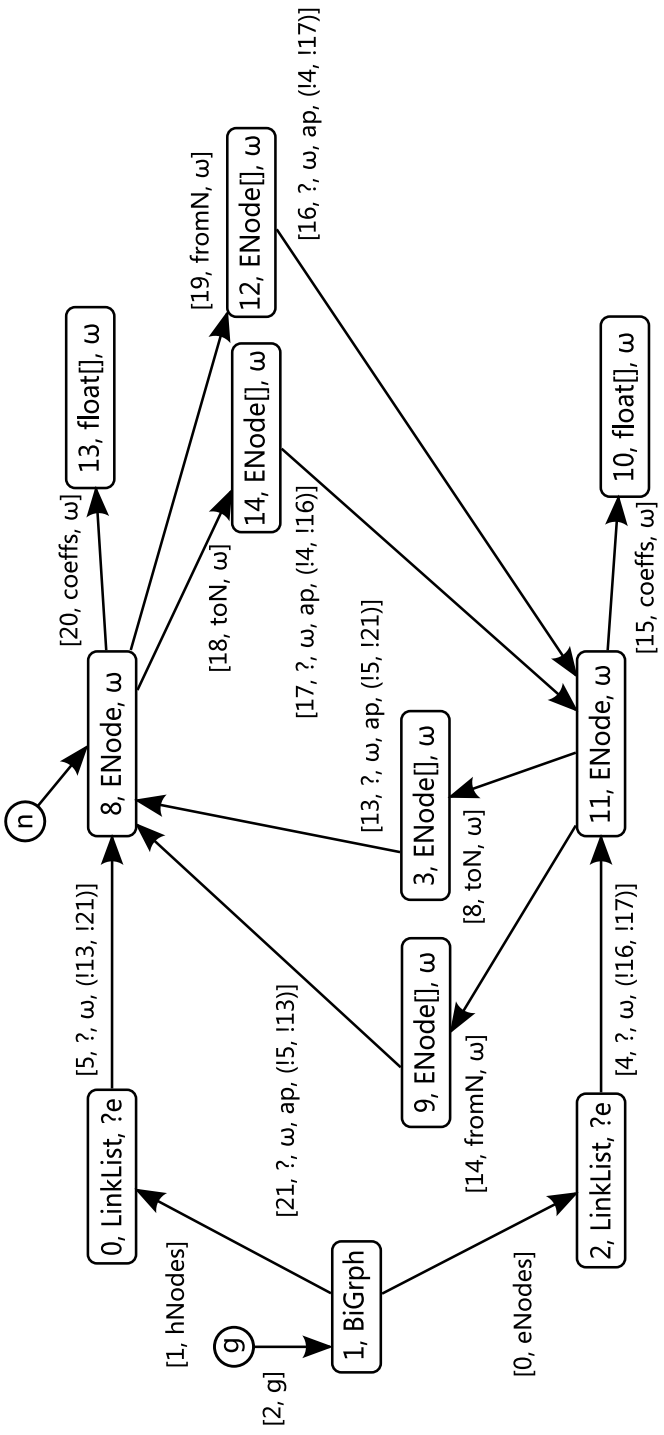


Figure 5.4: `computeNewValue`

```
static void computeNewValue(ENode n) {
  for(int i = 0; i < n.fromCount; i++)
    n.value -= n.coeffs[i] * n.fromN[i].value;
}
```

Figure 5.5: `Compute`

```
for(int i = 0; i < this.hNodes.size(); ++i)
  computeNewValue((ENode) this.hNodes.get(i));
```

Figure 5.6: Main Em3d Compute Loop

thus the loop can be unrolled and aggressively scheduled or can be vectorized.

The `computeNewValue` method is called from the loop shown in Figure 5.6 where we iterate through all the entries in the `LinkedList` updating the node values. Since we know that each reference in this collection refers to a unique `ENode` object (the edges with identifiers 4 and 5 have the non-interfere property) we know that each iteration of the loop updates a different `ENode` object. Thus, we can thread-parallelize the processing of this loop. Doing so results in a speedup of 3.21 on our quad-core test machine.

Finally, in this example we have two opportunities to improve the use of memory via static collection and to improve locality via collocation or dynamically updating the layout at key points. By looking at future reads/writes from fields we can determine that the field `toN` is never read from or written to again. Thus, we can collect this part of the heap prior to the main compute loop saving a substantial amount of memory (an array of size approx. 100+ entries per `ENode` object). Also by examining the later use of the data structures in the program we can determine that the lifetimes of the `ENode` objects is bounded by the lifetime of the `LinkedList` they are stored in. Thus we can either allocate the `ENode` objects inline with the `LinkedList` structure or prior to the compute loop we can dynamically reallocate the objects contiguously in memory.

5.2.4 Voronoi

Figure 5.7 shows the abstract heap that is computed at the end of the `buildDelaunay` method. This method performs a recursive construction of a voronoi diagram by building a diagram on the `VVertex` objects by computing a diagram for the left subtree, the right subtree and then merging the two diagrams (partial pseudo-code shown in Figure 5.8).

This particular heap example shows how the analysis uses a range of interfere properties to differentiate how various forms of sharing appear in a program. Figure 5.7 shows

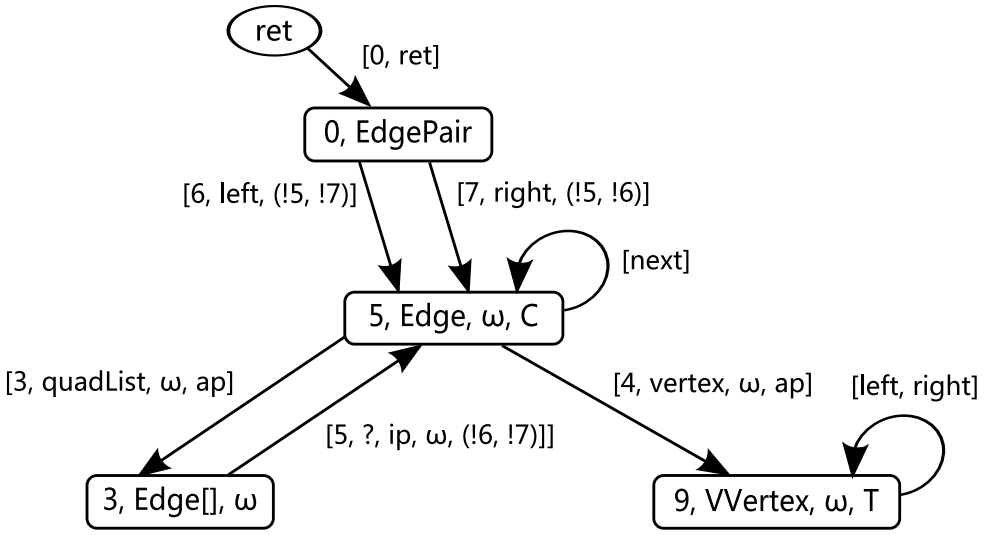


Figure 5.7: Voronoi

```
...
EdgePair delright = getRight().buildDelaunay(extra);
EdgePair delleft = getLeft().buildDelaunay(this);

retval = Edge.doMerge(delleft, delright);
...
```

Figure 5.8: `buildDelaunay` Pseudo-code

that at the return from the recursive `buildDelaunay` method the heap has a cyclic structure (based on the `next` pointer) of the edges in the triangulation (node 5). Each of these `Edge` objects has a reference to an `Edge[]` which contains pointers back into the set of `Edge` objects. Finally each `Edge` object has a pointer to a `VVertex` object which is laid out in a tree-like structure (the *(T)ree* property of node 9).

In addition to being able to precisely identify each of these structures on the heap and the connectivity relations between them, the analysis is able to compute interesting sharing information about how the pointers stored in the objects and arrays are related. If we look at edge 3, which represents the pointers stored in the `quadList` fields of the `Edge` objects, we see it has the *interfere* property *alias pointers (ap)* which means that two `Edge` objects may point to the same array. Similarly the edge abstracting the pointers stored in the `vertex` field (edge 4) also has the *interfere* property *ap* which indicates that the `VVertex` objects may be shared between the `Edge` objects (or that there may be pointers into the same tree structure).

More interesting is the edge (edge 5) from the node abstracting the `Edge[]` objects back to the node abstracting the `Edge` objects. This edge is labeled with the *interfere* property *interfering pointers (ip)* which indicates that the pointers stored in the arrays may point into connected parts of the cyclic structure but that none of them may alias (thus for all arrays $i \neq j \Rightarrow$ `Edge[i]` $\neq$ `Edge[j]`).

This sharing and structural information provide several opportunities for optimizing this particular program. The `doMerge` method performs many loop and conditional heavy modifications of `Edge` objects based on loads through the `Edge[]` arrays. Thus, we believe that the information that there are no aliasing pointers in these arrays would be very useful to standard load elimination, redundancy elimination, and scheduling optimizations. Unfortunately as we do not have these components in our optimization framework as of yet we are unable to measure the impact. However, we are able to use the structural separation information about the results of the `buildDelaunay` method and the tree recursive call structure to thread-parallelize the recursive call structure giving us a speedup of 2.43 on our test machine.

5.2.5 BH (Barnes-Hut)

Figure 5.9 shows the model that the analysis computes for the `hackGravity` method of the *Barnes-Hut* benchmark. The `bh` program performs a *fast-multipole* algorithm on the gravitational interaction between a set of bodies (the `Body` objects) and uses a space decomposition tree of `Cell` objects each of which has a `Vector` containing a subtree or a reference to the `Body` objects. The program also keeps two vectors for accessing the bodies, `bodyTab` and `bodyTabRev`. Figure 5.9 shows the state of the heap model after the loop body (Figure 5.10) that contains the majority of the computation in `bh`. This loop takes each `Body` object and walks the space decomposition tree (the `root` field) to determine a new acceleration value for the `Body` object (stored in the `newAcc` field).

Our analysis is not able to precisely resolve the construction of the space decomposition tree and conservatively assumes it may be a cyclic structure (shown by the `C` in node 17, representing the `Cell` objects). However, the analysis is able to determine that the `Cell` objects and the `Body` objects represent distinct regions in the program. The analysis is also able to determine a number of useful sharing properties with respect to how

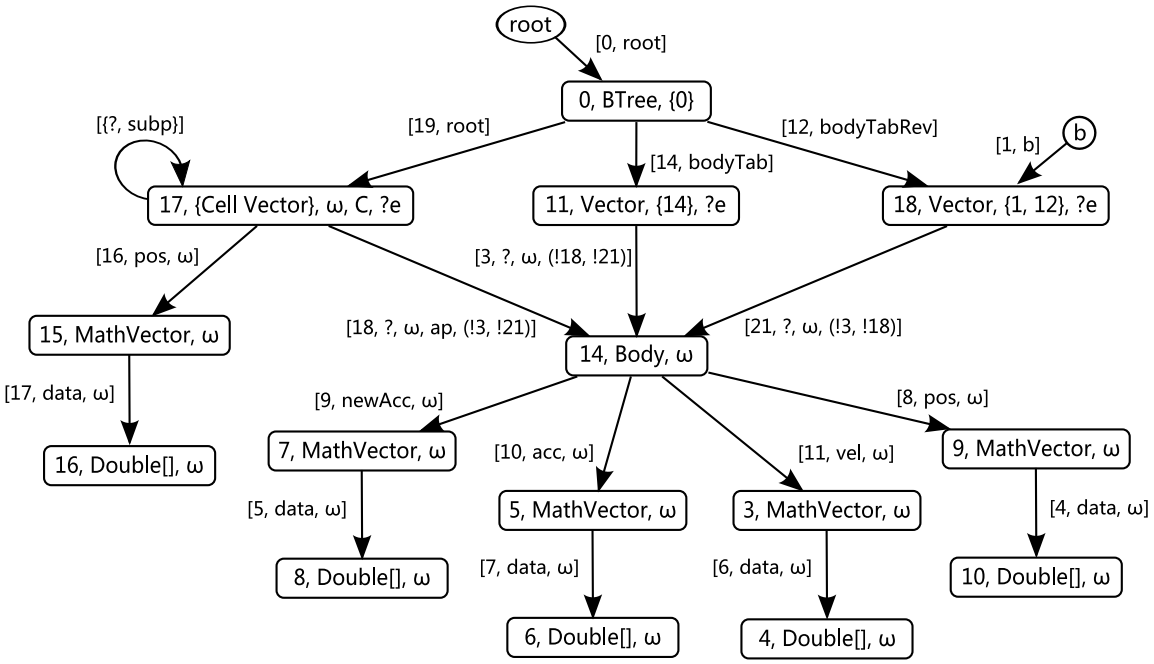

root
[0, root]
0, BTree, {0}
[{?, subp}]
[19, root]
[12, bodyTabRev]
[1, b]
b
[14, bodyTab]
17, {Cell Vector}, ω, C, ?e
11, Vector, {14}, ?e
18, Vector, {1, 12}, ?e
[16, pos, ω]
[3, ?, ω, (!18, !21)]
15, MathVector, ω
[18, ?, ω, ap, (!3, !21)]
[21, ?, ω, (!3, !18)]
14, Body, ω
[17, data, ω]
[9, newAcc, ω]
[8, pos, ω]
16, Double[], ω
7, MathVector, ω
[10, acc, ω]
[11, vel, ω]
9, MathVector, ω
[5, data, ω]
5, MathVector, ω
3, MathVector, ω
[4, data, ω]
8, Double[], ω
[7, data, ω]
[6, data, ω]
10, Double[], ω
6, Double[], ω
4, Double[], ω

Figure 5.9: BH

```
Iterator b = root.bodyTabRev.iterator();
while(b.hasNext())
  ((Body) b.next()).hackGravity(rsize, root);
```

Figure 5.10: Main Update, Gravity Computation

the `Body` objects (node 14) are stored in the two vectors (`bodyTab` and `bodyTabRev`) and the space decomposition tree. In particular it has assumed that the `Cell` objects in the space decomposition tree may have aliasing pointers (the *ap* entry in edge 18, marked in red if color is available) while there are no aliasing pointers stored in the `Vectors` `bodyTab` and `bodyTabRev`, the edges abstracting these pointers (edges 3, 21) have the *interfere* property *np* (the omitted default non-interfering value). We also see that the analysis has determined that the two vectors and the space decomposition tree all point to some of the same `Body` objects (the (!18, !21) in edge 3, (!3, !18) in edge 21 and (!3, !21) in edge 18 respectively these connectivity relations are marked in blue if color is available).

The sharing information about the pointers stored in each vector, the structural disjointness of the space decomposition tree and the `Body` objects combined with the observation that the space decomposition tree is only read in the loop body and that the only part of the heap which is modified is never read (the `newAcc` field) is sufficient to ensure that there is no heap-carried dependence in this loop. Thus, we can safely thread-parallelize the loop body, achieving a factor of 3.09 speedup on our quad-core test machine.

Of interest from a memory management standpoint are the `MathVector` objects which are used to represent k-dimensional points in space or vectors (a small `static final int` known at compile time). Examining the heap through the entire program shows that each `MathVector` points to a unique array at all times, the edges from the `MathVector` objects to the arrays are always *non-interfering (np)* (the omitted default value), and the array nodes never have multiple edges from different `MathVector`

nodes. This allows us to safely inline the elements in the arrays directly into fields in the `MathVector` objects (in effect giving the arrays value instead of reference semantics). This has the beneficial effect of increasing the data locality and replacing several loops with straight line code, resulting in a 23% reduction in the runtime of the single-threaded program, as well as reducing the size of the `MathVector`/Array composite structure object by a pointer (and the overhead of an array), resulting in a 37% reduction in memory usage.

Chapter 6

Normalization

This chapter introduces a normal form for the *labeled storage shape graphs*. The normal form serves two functions in the analysis: it allows for two *labeled storage shape graphs* to be efficiently compared (Chapter 7) and it ensures that there are a bounded number of *labeled storage shape graphs* that are possible at each control flow join point (Chapter 9). These two properties are needed to ensure the termination (termination of the analysis is guaranteed for a finite domain) and efficiency of the analysis. We note that the normal form operation may reduce the precision of the graph model (it acts as a pseudo-widening [17]). While this potentially reduces the precision of the analysis the result is still a safe approximation of the program.

There are many possibilities for satisfactory normal form definitions and the normal form need not be unique (and the operation to compute it need not even be a function). The only requirement is it must ensure the existence of a fixed bound on the number of nodes/edges that can appear in a graph.

Definition 16 (Normal Form Requirements)**.** *The normal form must satisfy:* $\exists N_{\max}, E_{\max} \in \mathbb{N}$ *s.t.* $\forall g$ *where* g *is in normal form:*

1. $|\{\textit{node } n \mid n \in g\}| \leq N_{\max}$.

2. $|\{\textit{edge } e \mid e \in g\}| \leq E_{\max}$.

Since the objective of constructing this finite set is to ensure that the analysis terminates in a reasonable amount of time, when analyzing a given program P_r we do not need to construct a universal normal form and bound for all programs, but can instead perform the construction parametrically with respect to the types used in the program P_r.

6.1 Simple Normal Form Definition

We begin with a simple definition for a normal form operation that satisfies the above definition. While this normal form operation is surprisingly effective in many cases, there are a number of problems with the definition that result in excessive losses of accuracy when using it in practice. We examine the causes of these precision losses and amend the definitions to avoid them in Section 6.2 and 6.3.

We can identify all the types in a program that may be recursive by looking at the type graph for the program. Below we define the construction of a type graph for a simple type system without interfaces or inheritance. This construction can be easily extended to handle object-oriented language features.

Definition 17 (Static Program Type Graph)**.** *For a given program P_r we can use the declared types $\{\tau_1, \ldots, \tau_k\}$ in the program to construct a type dependence graph as follows:*

1. *For each $\tau_i \in \{\tau_1, \ldots, \tau_k\}$ we add a node to the graph.*

2. *If τ_i has a field of type τ_j we add an edge (τ_i, τ_j) from the node for τ_i to the node for τ_j.*

Based on this construction we can identify types that are recursive (based on the static type information) as the types that are contained in the same strongly connected component of the graph or that have self edges in the graph.

Definition 18 (Statically Recursive Types)**.** *For a given program we can use the declared types to compute which objects may be a part of the same recursive structure, given types τ, τ' we use the following terminology:*

1. τ, τ' *are* statically recursive *iff in the* Static Program Type Graph *($\tau \neq \tau' \wedge \tau, \tau'$ are in the same strongly connected component) $\vee$ ($\tau = \tau'$ and there is a self edge (τ, τ)).*

2. τ *is a* statically recursive *type iff $\exists \tau'$ s.t. τ, τ' are* statically recursive.

Definition 19 (Simple Normal Form)**.** *A graph g is in the simple normal form if:*

- $\forall$ *nodes n, n is reachable from a variable node.*

- $\forall$ *nodes n, n' s.t. $\exists$ edge e from n to n' then ($\nexists \tau \in n$.*types*, $\tau' \in n'$.*types *s.t. τ, τ' are* statically recursive*).*

- $\forall$ *nodes n $\nexists$ edges e, e' both starting at n where $e \neq e' \wedge e$.*offset $= e'$.offset*.*

Given the operations to combine nodes/edges in Section 6.5 it is straightforward to transform an arbitrary heap graph into its *simple normal form*. Further it is clear that the simple normal form satisfies our requirements as (1) the max depth in the graph is bounded by the size of the largest statically non-recursive structure since all recursive structures are represented by a single node (no two nodes with an edge connecting them can contain *statically recursive* types) and (2) the maximum branching factor for any node is limited by the number of fields declared in the program.

6.2 Equivalent Edge/Node Identification

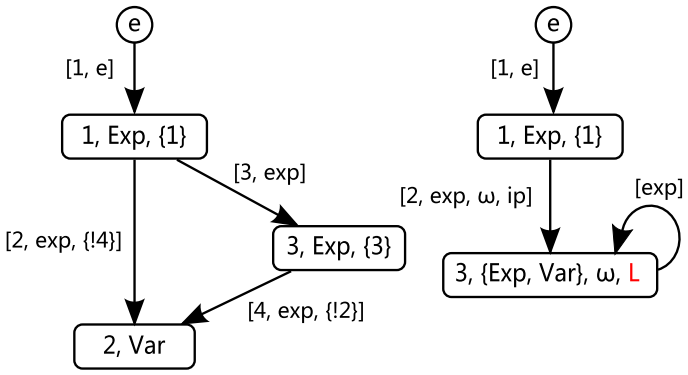
To allow us to relax the condition on the outgoing edges for a given node we want to introduce a notion of *equivalence* of two nodes/edges that captures our intuition of when two nodes n, n' abstract similar regions of the concrete heap. Since the *equivalence* predicate is used to determine the maximum number of out edges each node may have, we can improve efficiency by minimizing the number of equivalence classes created by this relation. The tradeoff between precision and performance that we have found to be acceptable is determined by the following conditions: (1) are all the types represented by the nodes non-recursive (or may both nodes represent recursive types) and (2) what variables can reach the objects in the regions abstracted by the nodes?

Definition 20 (Recursive Similarity)**.** *Given nodes n, n' and the* statically recursive *type information, n, n' are* recursive similar *iff either of the following holds:*

1. *($\exists \tau \in n$.types, τ is statically recursive) $\wedge$ ($\exists \tau' \in n'$.types, τ' is statically recursive)*
2. *($\forall \tau \in n$.types, τ is not statically recursive) $\wedge$ ($\forall \tau' \in n'$.types, τ' is not statically recursive)*

Thus, the nodes are *recursive similar* if either they both abstract all non-recursive types or they both may abstract an object with a recursive type. An example of why this is important is the common construction of k-ary trees using arrays to hold either a recursive subtree or a non-recursive leaf object, which is often shared by multiple nodes.

Consider a simple program which has two types, `Exp` and `Var`, where the type `Var` is a non-recursive leaf type and `Exp` has a field, `exp` which either points to a `Var` object or another `Exp` object. Figure 6.1(a) shows an abstract heap representing a small abstract heap structure using these types. If we were to group the recursive and non-recursive nodes together we get the result in Figure 6.1(b) which has been forced to create a List



(a) Recursive Similarity (b) Without Recursive Similarity

Figure 6.1: With and Without Recursive Similarity

structure when merging the `exp` edges. This is a substantially less precise description of the program state and as the analysis continues will result in the overly conservative approximation that the structure is a DAG instead of the desired discovery that we have a recursive list (or tree) with shared leaf objects. Conversely with the use of *recursive similarity*, the grouping will keep the edges distinct and leave the original heap unchanged. The `BH` program discussed in Chapter 5 shows a larger example of where this property allows the model to represent the `Body` leaf objects as a distinct region from the `Cell` objects that make up the space decomposition tree.

Reference Similarity. If we have two nodes n, n' and the objects abstracted in the region by n are all stored in an array A and all the objects in the region abstracted by n' are stored in array A and a second array B then it is reasonable to assume that the programmer has partitioned these objects differently for some reason. Thus, we want to preserve this information by keeping the nodes distinct, we show this situation in Figure 6.2. We can ensure that the information on which collections and variables refer to which sets of objects is maintained by using the following definition of *reference similarity*.

Definition 21 (Reference Similarity)**.** *We say two nodes n, n' are reference similar if given the set of in edges to n, $E_{\text{in}} = \{e_1^n \ldots e_k^n\}$, the set of in edges to n', $E'_{\text{in}} = \{e_1^{n'} \ldots e_k^{n'}\}$, and the*

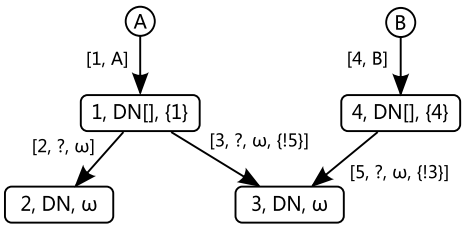


Figure 6.2: Not Reference Similar (based on variable reachability)

set of variables that can reach node n, $V_r = \{v_1^n \dots v_r^n\}$*, the set of variables that can reach node* n'*,* $V'_r = \{v_1^{n'} \dots v_s^{n'}\}$*, the following holds:*

$$\{e.\text{offset} \mid e \in E_{\text{in}}\} = \left\{e'.\text{offset} \mid e' \in E'_{\text{in}}\right\} \wedge V_r = V'_r$$

This definition ensures that if two nodes are treated differently with respect to the types of objects they are stored in or the variables that reach them then they are kept separate.

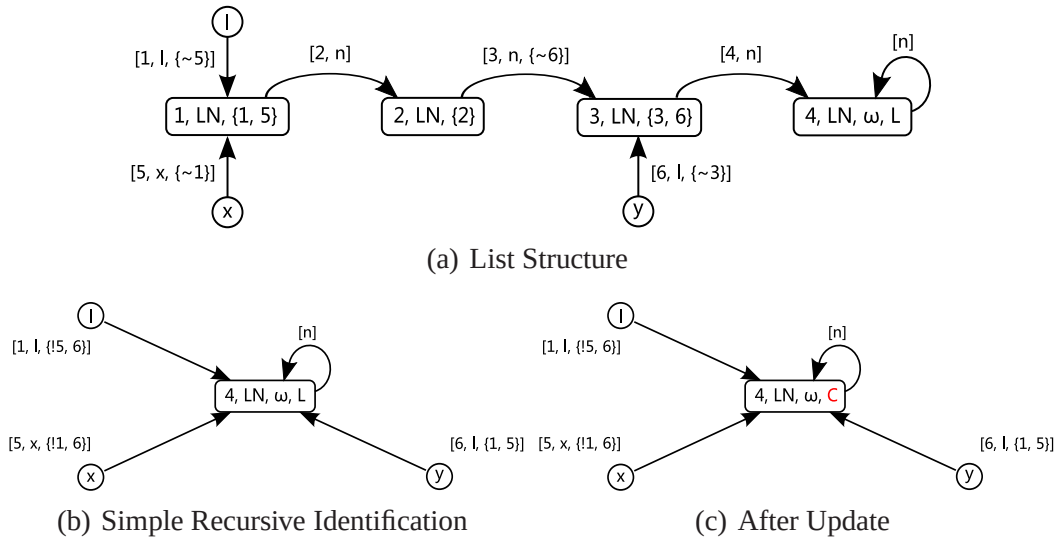
Definition 22 (Equivalent Nodes/Edges)**.** *Given the above definitions we define* edge equivalence. *Given a node n and two out edges* e, e' *which start at node n and end at nodes* n_e *and* $n_{e'}$ *respectively we say* e, e' *are* equivalent *if:*

$$(e.\text{offset} = e'.\text{offset}) \wedge (n_e, n_{e'} \textit{ are } \text{recursive similar}) \wedge (n_e, n_{e'} \textit{ are } \text{reference similar})$$

6.3 Recursive Components

In this section we define a predicate that takes two nodes (n, n' where $n \neq n'$) with one or more edges between them and determines if they may be part of the same recursive component of a data structure.

A Simple Initial Approach. Using the static type information to determine which parts of the *labeled storage shape graph* may represent recursive structures can cause massive



(a) List Structure

(b) Simple Recursive Identification

(c) After Update

Figure 6.3: Result of Simple Recursive, on List Remove

losses in precision during the analysis. Consider the case of removing an element from a linked list where we have multiple variables pointing into the same list structure, and maintaining the order relation between them is critical to accurately modeling the final removal of an element. Using the above approach will result in the loss of all order information between the variables in the list. Figure 6.3(a) shows one of the abstract heap graphs that arises during the analysis of the `listRemove` method. The result of the simplistic recursive component elimination is shown in Figure 6.3(b) and results in the loss of all order relations between `x` and `y`. The update `x.next = y` then results in the analysis (very conservatively) assuming that the region may become cyclic as shown in Figure 6.3(c).

Safe Nodes. To avoid this problem we introduce the notion of *safe nodes*. We say a node is safe if it represents an interesting point in a recursive data structure (a point where the program is accessing a specific node in the data structure though a variable reference, as in the above example, or a non-recursive data structure pointing into specific locations in the recursive structure) and we always keep these nodes separate/distinct from any other recursive components of a data structure.

If we have a recursive data structure and we store references to important points in it via another data structure we want to be able to maintain the relations between these specific points in a data structure for when they are accessed later in the program. This is a generalization of maintaining the precise locations of variable references in a recursive data structure. This particular expansion is important to analyzing situations of the form: a method returns a pair of nodes in a list and we want to remove all the elements in the list between the `first` and `second` entries of the pair. If the analysis does not maintain the order relation between the targets of the two fields in the non-recursive pair object in the recursive list structure we cannot accurately model the effects of the remove operation (e.g., after normalization we would be forced to conservatively assume that the target of the `second` field could come before the target of the `first` field).

Definition 23 (Safe Node)**.** *A node n is* safe *if it is a* (S)ingleton *node and either of the following hold:*

1. $\exists$ *variable v that refers to n.*
2. $\exists$ *edge e s.t. e starts at a node* n_s *where* $\forall \tau_s \in n_s.\text{types}, \tau \in n.\text{types}$, τ_s, τ *are not statically recursive).*

Online Computation of Recursive Structures. The detection of potentially recursive types by examining the type graph of the program can lead to overly aggressive merging of components of a heap graph. Consider a program with the object types τ_1, τ_2, τ_3 which are mutually recursive on the `n` field. If we have the abstract heap graph in Figure 6.4 we can see that the 2^{nd} and 3^{rd} nodes in the list are statically recursive according to the definitions above but that there is no complete recursive structure present in the program (no type appears multiple times in the same structure). This can occur frequently with pre 1.5 Java collections as their contents are untyped (they may contain any type of object) and are thus statically recursive with all other types.

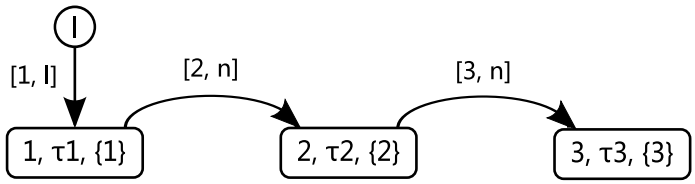


Figure 6.4: Recursive Types But No Complete Structure

To avoid this problem we perform an online detection of the recursive structures in a program so that instead of saying n, n' are recursive if there is an edge from n to n', neither n, n' is *safe* and $\exists t \in n.types, t' \in n'.types$ s.t. t, t' are statically recursive, as in our initial simple approach we use a reachability construction based on the abstract heap structure.

Definition 24 (Online Recursive Nodes)**.** *We definition of n, n' as being online recursive if:*

$\exists$ *an edge from n to n', neither n, n' is* safe *and $\exists n_\tau$ s.t. there is a path (possibly empty) from n' to n_τ that does not pass through any* safe *nodes and* n.types $\cap$ n_τ.types $\neq \emptyset$.

This ensures that if we identify two nodes as being recursive then they are part of a complete (we have a least one complete cycle from a node of a given type to another node of the same type) and unbroken (there are no *safe* nodes in this cycle) recursive structure in the program.

Recursive vs. Back Pointers. Many programs use back pointers causing the above definition to identify any cyclic structure as recursive (since trivially we can let $n_\tau = n$). This causes the grouping of cycles in the graph into single nodes with the *layout (C)ycle*, which can lead to substantial imprecision. Figure 6.5 shows an example of such a heap. We can see that even though the heap structure is finite, the back edge will cause our recursive component definition to group the 2^{nd} and 3^{rd} nodes into the same recursive component.

To address this problem we modify the recursive definition slightly to ignore back

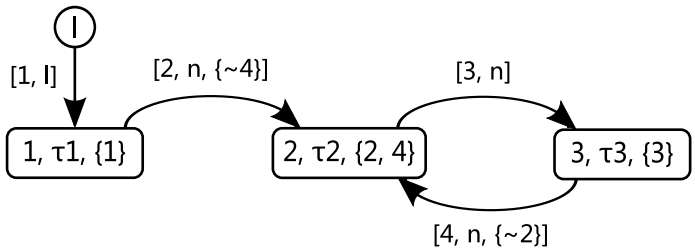


Figure 6.5: Recursive Cycle

pointers when computing if two nodes are recursive. This gives us the final definition for recursive nodes.

Definition 25 (Recursive Nodes)**.** *Given the function* depth *which returns the depth of a node in the graph, nodes n, n' (where $n \neq n'$) are* recursive *if:*

$\exists$ *edge e from n to n', neither of n, n' is* safe *and $\exists n_\tau$ s.t. there is a (possibly empty) path $\psi_r \langle (n_1^s, n_1^e) \ldots (n_k^s, n_k^e) \rangle$ from n' to n_τ s.t. $\forall (n_i^s, n_i^e) \in \psi_r, \text{depth}(n_i^s) < \text{depth}(n_i^e)$ (where* depth *is the depth of the node in the graph), $\forall (n_i^s, n_i^e)$, neither n_i^s or n_i^e is safe and, n.types$\cap$ n_τ.types $\neq \emptyset$.*

6.4 Focus Operation

The focus operation is a set of rewrite rules for the nodes/edges in a graph that we use to propagate the implications of the various node/edge properties and the structure of the graph. Given a node *n*, a set of incoming edges $E^i = \{e_1^i, \ldots, e_m^i\}$ and a set of outgoing edges $E^o = \{e_1^o, \ldots, e_n^o\}$ the *focusNode* operator updates the node and edges using the following rewrite rules:

$$\frac{n.linearity = 1 \wedge \exists e \in E^i, e.interfere \neq ap}{e.linearity \leftarrow 1}$$

This rewrite rule allows us to determine that if a node represents a single object then if an edge represents a set of pointers that do not alias then the set must only contain a single

pointer (since all pointers in this set must point to the single object in the region abstracted by the node).

$$\frac{n.linearity = 1 \wedge \exists e \in E^o, e.offset \notin \{\texttt{?, bi, ai}\}}{e.linearity \leftarrow 1 \wedge e.interfere \leftarrow np}$$

This inference rule allows us to determine that if a node represents a single object then all the out edges which represent pointers stored in the fields of the object (i.e. pointers not stored in summary containers such as arrays or collections) must represent a single pointer (and naturally a single pointer cannot share with itself so the edge is non-interfering).

$$\frac{n.layout = S}{n.types \text{ only contains type names consistent with the incoming edge types}}$$

This rule states that if there are no internal pointers in the region abstracted by the node (a Singleton layout) then each object must be referred to by a reference abstracted by one of the incoming edges. Thus the set of types that the objects may be is limited by the types of the targets of the references abstracted by the incoming edges.

$$\frac{\exists e \in E^i, e.linearity = 1 \wedge e \text{ dominates } n}{n.linearity \leftarrow 1}$$

This rule states that if we have an incoming edge which represents a single pointer and every object in the region abstracted by the target node is referred to by a pointer in this edge then there is only a single object in the region.

$$\frac{E^i = \{e\} \wedge n.layout = S \wedge \text{ start node of } e \text{ has } linearity\ 1}{n.domset = \{e\}}$$

This rule states that if there is a single incoming edge to a node that abstracts a region of the heap with no internal pointers then every live object in this region must be pointed to by one of the references in the incoming edge (and to ensure we don't later introduce spurious dominance information the start node of the edge must represent a single object).

$$\frac{\exists e_x, e_y \in E^i \text{ s.t. } e_x \text{ dominates } n \wedge e_y \text{ dominates } n}{e_x \text{ domEQ } e_y}$$

This inference rule allows us to determine that if every object in a given region are pointed to by a reference abstracted by e_x and every object is also pointed to by a reference abstracted by e_y^i then the set of objects pointed to by references abstracted by e_x equals the set of objects pointed to by references abstracted by e_y.

$$\frac{E^i = \emptyset}{\text{then remove } n \text{ and all the edges in } E^o \text{ from the graph}}$$

This rule ensures that if the region represented by the node is unreachable by the program then we remove the node and all out edges since they are semantically meaningless.

6.5 Normal Form

Given the definitions for *equivalent edges* and *recursive nodes* which provide the required heuristics for determining which nodes should be grouped into the same recursive components and which nodes represent similar regions on the concrete heap, we can define our improved normalization method. We begin by defining how to compute a summary node/edge that safely approximates the properties of a given set of nodes/edges. The most general way to do this would be to compute a single summary node/edge over the entire set. However, this is a difficult problem in the general case and we opt for the simpler approach of processing the nodes/edges in a pairwise fashion.

6.5.1 Node Summarization

When summarizing two nodes, n_a and n_b, there are three possibilities. The first is that there are no edges between the nodes, there are only edges in one direction between nodes (from n_a to n_b or n_b to n_a, but not both) and when there are edges from n_a to n_b and from n_b to n_a.

If there are no edges between the nodes we use the *mergeNoEdge* method to compute the summary representation. This method is mainly a component-wise operation (in that we take the max of the properties or union of the relations). The two only non-trivial details are to set the *linearity* of the summary node to the sum ($\widetilde{+}$) of the *linearity* of the two nodes, which is always ω (since the summary node may represent multiple objects) and to clear the out edges from any dominance relations. The clearing of the dominance information is key to ensuring that later writes that might make objects abstracted by this node unreachable (and this potentially affecting the dominance properties) are correctly handled (by conservatively forgetting the dominance properties of any summary nodes). The *mergeEdgeOneWay* operation on a pair of nodes that have connecting edges is more complicated. In particular we need to account for the fact that the edge(s) connecting nodes n_a and n_b will affect the layout and the internal connectivity of the new summary node.

Algorithm 1: mergeOneWay

input : graph g, n_a, n_b nodes, *ebt* set of edges from n_a to n_b

n_a.types $\leftarrow$ n_a.types $\cup$ n_b.types;

n_a.linearity $\leftarrow$ ω;

n_a.layout $\leftarrow$ combineLayout(n_a.layout, n_b.layout, *ebt*);

n_a.connR $\leftarrow$ combineConnR(n_a.connR, n_b.connR, *ebt*);

n_a.nodeDom $\leftarrow$ $\emptyset$;

n_a.activeIter $\leftarrow$ $\perp$;

n_a.empty $\leftarrow$ unknown;

n_a.atFirst $\leftarrow$ false;

remap all edges incident to n_b to be incident to n_a;

foreach *out edge e* **do**
    if *e.offset* $\in \{\text{ai}, \text{at}, \text{bi}\}$ **then** *e.offset* $\leftarrow$?;
    remove *e* from any dominance relations;
deleteNode(g, n_b);

The algorithm *combineLayout*(l_a, l_b, ebt) (Algorithm 2), is based on a case analysis of the internal layout that results from the possible combinations of layouts for n_a, n_b along with the total number of pointers represented by *ebt* and the potential that any pointers in the edges represented by *ebt* interfere. We enumerate the possible combinations of the *ebt* edges and the layout types. Then for each case we use the semantics of the edge and layout properties to determine the most general layout type that may result from this particular case. For example if we have two *(S)ingleton* nodes connected by an edge of *linearity* 1 then the most general *layout* for a node that summarizes these nodes and the edge is a *(L)ist*.

Algorithm 2: combineLayout

input : l_a, l_b layout types, *ebt* set of edges from n_a to n_b

output: the layout of the combined node

$mayInterfere \leftarrow \bigvee\{e \in ebt | e.interfere = ip \vee ap\}$;

$totalPointers \leftarrow \sum\{e \in \text{ebt} | e.linearity\}$;

$notSingletons \leftarrow s_a \neq Singleton \wedge s_b \neq Singleton$;

$isDAGgraph \leftarrow totalPointers > 1 \wedge notSingletons$;

$l_r \leftarrow \max(l_a, l_b)$;

case *(mayInterfere* $\vee$ *isDAGgraph)* **return** $max(l_r, MultiPath)$*;*

case $(l_a = List)$ **return** $max(l_r, Tree)$*;*

case $(l_a = l_b = Singleton)$ **return** $max(l_r, List)$*;*

otherwise return l_r*;*

The *combineConnR* function updates the internal connectivity information in n_a to reflect that it now represents the combined regions for n_a and n_b. This involves computing the binary connectivity relation for all the edges that are incident to the new summary node based on the connectivity information in the argument nodes n_a, n_b, and the edges that connect the argument nodes, *ebt*.

$$combineConnR(n_a, n_b, ebt) = \{(e, e', n, c) | (e, e', n, c) \in g.connR \wedge e \notin ebt \wedge e' \notin ebt\} \cup$$

$\{(e, e', n, ip) | \exists e_b \in ebt \text{ s.t. } (e, e_b, n_a, ip|ap) \in g.connR \wedge (e_b, e', n_b, ip|ap) \in g.connR\}$.

To merge two arbitrary nodes n, n' we use Algorithm 3 which selects the appropriate method for merging two nodes based on the existence of edges between them.

Algorithm 3: mergeNode

input : node n, n', graph g

output: None

if $\exists$ *edges from n to* n' *and* n' *to n* **then**
    replace n, n' with the top node $\top_n$;

else if $\exists$ *edges from n to* n' **then**
    mergeOneWay(g, n, n');

else if $\exists$ *edges from* n' *to n* **then**
    mergeOneWay(g, n', n);

else
    mergeNoEdge(g, n', n);

6.5.2 Edge Summarization

The edge *merge* method is only well defined when two edges start at the same offset in the same node and end at the same node. The method checks the end connectivity information to determine how the component abstraction should be combined. If the edges *share* then the pointers that these edges represent may alias and we set the summary edge as *ap*, similarly if the edges are *connected* then the pointers that these edges represent may interfere and we set the summary edge as *ip*, otherwise they are *non-interfering*. Once we have computed the relation property between the pointers in the two edges we take the max of this property and the *interfere* properties of the two edges. For the rest of the components that are used to represent an edge, we can simply combine them component-wise. The edge join algorithm uses the function *updateInternalConnInfoEdgeJoin*(n_s, n_e, e_a, e_b) to update the internal connectivity info in n_s and n_e to represent the fact that e_a now repre-

sents pointers from e_a and e_b and thus for each connectivity relation that holds for either e_a or e_b should now hold for e_a.

Algorithm 4: mergeEdge

input : g graph, e_a, e_b edges, n_s, n_e the nodes, e_a, e_b start and end at

$e_a.linearity \leftarrow \omega$;

if e_a, e_b share **then**
    $e_a.interfere \leftarrow ap \sqcup e_a.interfere \sqcup e_b.interfere$;

else if e_a, e_b connected **then**
    $e_a.interfere \leftarrow ip \sqcup e_a.interfere \sqcup e_b.interfere$;

else
    $e_a.interfere \leftarrow e_a.interfere \sqcup e_b.interfere$;

updateInternalConnInfoEdgeJoin(n_s, n_e, e_a, e_b);

deleteEdge(g, e_b);

6.5.3 Normalization

We begin by giving a definition for when a graph is in normal form. Then we define a helper method to remove all equivalent edges from a given node and finally present the full normalization operator.

Definition 26 (Labeled Storage Shape Graph Normal Form)**.** *A graph g is in normal form if:*

- $\forall$ *nodes n, n is reachable from a variable node.*
- $\forall$ *nodes n, n cannot be* refined *into disjoint nodes, see Section 8.2.*
- $\forall$ *nodes n, none of the focus operations can be applied.*
- $\nexists$ *nodes* n, n' *s.t.* n, n' *are* recursive.
- $\forall$ *nodes n* $\nexists$ *edges* e, e' *both starting at n where* $e \neq e' \wedge e, e'$ *are* equivalent.

Equivalent Edge Removal. To remove all the *equivalent edges* from a node we check if there are two distinct edges that have the same *offset* and their targets are either the same node or are *node equivalent*.

Algorithm 5: removeEquivalentEdges

input : node n, graph g

output: None

while $\exists$ *equivalent edges* e, e' **do**
    $n \leftarrow$ endpoint of e;
    $n' \leftarrow$ endpoint of e';
    if $n \neq n'$ **then** $mergeNode(g, n, n')$;
    $mergeEdge(g, e, e')$;

Normalize Graph. The full algorithm for normalizing the heap graph is shown in Algorithm 6. For efficiency we begin by splitting any nodes that represent disjoint regions into multiple nodes (see Section 8.2), then we apply the refinement rules to all the nodes, these operations reduce the number of possible normal forms for a given graph. Next we find and remove any *equivalent edges* and finally we merge all the recursive nodes in the graph. This process is repeated until the heap graph is no longer changing.

Example Normalization. In Figure 6.6 we show an abstract heap graph and the resulting normalization of the graph. The graph shown in Figure 6.6(a) has an array containing several `DN` nodes, one of which is referred to by the variable `x`, and a list of `LN` nodes. While the list of `LN` nodes was the variable `y` pointing into an interior point in the list.

In Figure 6.6(b) we have applied a number of the *focus* rules to propagate information in the model. For node 3 we have used the third rule to infer that the since the *linearity* of the in edge is 1 then the *linearity* of the node is also 1. In node 7 we have used the type consistency rule to determine that the only valid type for the node is `LN`. Also for edges 6,

Algorithm 6: normalizeGraph

input : graph *g*

output: None

Remove all unreachable nodes from *g*;

while *g is changing* **do**

  while ∃ *node n s.t. there is a partition of the in edges* **do**
    g.refineDisjointEdges(*n*);

  while ∃ *node n s.t. n can be focused* **do**
    focus(*n*);

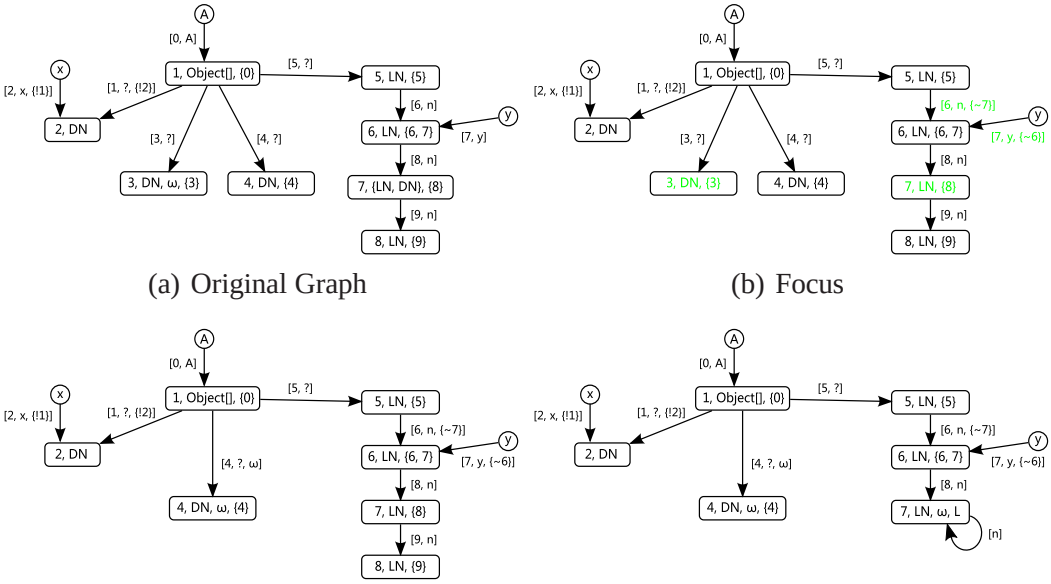
  while ∃ *node n s.t. n has* equivalent edges **do**
    removeEquavalentEdges(*n*, *g*);

  while ∃ *nodes* n, n' *that are recursive* **do**
    mergeNode(*g*, *n*, n');

7 we used the sixth rule to infer that since both edges dominate node 6 then they must be dominance equal.

Figure 6.6(c) shows the heap model that results from removing all equivalent edges. Since edges 3, 4 are equivalent (they are at the same offset and the nodes are *reference* and *recursive* similar) we have replaced them with a single summary node (node 4). This node represents many objects of type DN and since the original nodes were both *(S)ingleton* layout and there were no edges between them the resulting summary node is also a *(S)ingleton* layout. Similarly the edges were merged (into edge 4) and since they were *disjoint* the resulting summary edge is *non-interfering*. Nodes 2 and 5 are not equivalent with node 4 since they are not *reference* similar and not *recursive* similar respectively.

The final Figure 6.6(d) shows the result of applying the recursive node elimination (and is also the final result of the normalization). In this figure we have merged nodes 7, 8 into a single summary node (7). Since there was a single edge of *linearity* 1 connecting them and they were both *(S)ingleton* nodes the summarization of the regions the represent is a region with a *(L)ist layout*. Node 6 has not been merged into the recursive list structure



(a) Original Graph (b) Focus

(c) Equivalent Edges (d) Recursive Nodes

Figure 6.6: Normalization

since it is *safe* according to our definition of recursive nodes.

Chapter 7

Domain Order and Combine

Given our abstract heap domain ($\hat{H}$) (*labeled storage shape graphs* from Chapter 3) and the domain of equivalent values ($\widetilde{U}$) (the *must equal* domain from Section 4.2) we are ready to construct the full domain $\wp(\widetilde{U} \times \hat{H})$ used for program analysis. This construction is done in several steps. First, in Section 7.1, we construct a partial order on the *properties* that we use to label the nodes/edges of the *labeled storage shape graphs* and a way to compute a safe upper approximation ($\tilde{\sqcup}$) of two *property* values. Then, in Section 7.2, we define a partial order operation on a subset of the *labeled storage shape graphs* and a specialized upper approximation operator for (under certain conditions) merging two *labeled storage shape graphs*. Finally, in Section 7.3, we define the full domain, a partial order ($\leq_f$), and upper approximation ($\tilde{\sqcup}_f$) operations for the domain that is used to do the program analysis. We conclude with an outline of the proof that the resulting order operation and abstract domain provide a safe approximation of the concrete program states, $\sigma \leq_f \sigma' \Rightarrow \gamma_f(\sigma) \subseteq \gamma_f(\sigma')$, where γ_f is the concretization operation.

Many of the operations we define is this section are parametrized based on the existence of a subgraph isomorphism Π between two *labeled storage shape graphs* g, g' ($\Pi : g \mapsto g'$). For simplicity we assume this map is composed of multiple submaps that

relate edges, nodes, the *connR* relation and the *domEQ* relation. Thus we can apply it to the entire graph g or any of the individual subcomponents.

One major concern with this use of subgraph isomorphism in the follow computations is the computational complexity of the problem (which is known to be *NP-Complete*). Fortunately our graph domain provides us with a substantial amount of extra information that can be used to make the problem nearly linear in practice (although the worst case runtime is still exponential). First is the existence of a class of root nodes, the local and static variable names, that can always be directly matched. These initial matches provide a starting point that we can use to extend the subgraph isomorphism in a breadth-first manner. Secondly we can use the edge labels to quickly discard most infeasible isomorphisms. That is, in all of the operations described here we are only interested in isomorphisms that are consistent with the edge *offsets*. Thus we know that for edge $e \in g$ only edges $e' \in g'$ s.t. $e.offset = e'.offset$ are feasible and can discard all other possible matches. Along with the standard subgraph isomorphism heuristics this additional information from the root variables and the edge labels allows us to compute the isomorphisms with an average of less than one backtrack choice point per subgraph isomorphism computation done in our benchmarks.

7.1 Order and Join on Label Properties

In Chapter 3 we defined a number of instrumentation properties as labels for the labeled storage shape graphs which filter the possible concretizations for a given abstract heap graph. The definitions of these properties were designed to induce a natural order on them (based on implication in terms of the concrete heap predicates that were used to define them). For each of the instrumentation properties we first defined a number of predicates in first-order logic and graph reachability and then each of the instrumentation properties were defined in terms of which of these predicates *may*/*must* hold. This construction

induces a natural order on the instrumentation properties based on the sets of concrete predicates that they admit in a concretization which in turn implies the abstract domain is a safe approximation of the concrete domain. We will return to this later after we have fully defined the order operation in the abstract domain.

For each of the properties we define an order operation $\leq$ and an upper approximation operator $\tilde{\sqcup}$. For the order operation to be safe with respect to the concrete program heaps we must have that for any label values p, p' where $p \leq p'$ if we replace an occurrence of p in one of the labels of the abstract graph g with p' to get g' then $\gamma(g) \subseteq \gamma(g')$. As described above this property is trivially satisfied by the construction of the label properties based on satisfaction of subsets of concrete heap properties. In order to ensure that the fixpoint computation is safe, each pair of definitions must ensure that $p \leq p \tilde{\sqcup} p'$ and $p' \leq p \tilde{\sqcup} p'$, where p, p' are values from the property set we are defining $\leq$ and $\tilde{\sqcup}$ on.

Offsets. For the offsets the analysis uses a strict notion of identity so that two edges are only matched in the subgraph isomorphism if they have the same *offset* and we never try to combine two edges with different *offset* labels.

Types. The *type* and *offset* properties are handled in the usual manner. For the *type* property the order is subset inclusion on the type labels and the upper approximation operation is set union which is consistent with our definition of the abstract *type* property where if a given type name is in the set the concretization of the node may include objects of the given type.

Definition 27 (Type Property Operations)**.** *Given type sets t, t':*

- $t \leq t' \Leftrightarrow t \subseteq t'$.
- $t \tilde{\sqcup} t' = t \cup t'$.

Linearity. The linearity property is associated with both edges and nodes and ranges over the domain of labels $\{1, \omega\}$. Based on our definition we know that the value 1 implies the concretization of the node/edge must produce 0 or 1 objects/references and that the value ω implies the concretization may contain any number of objects in the range $[0, \infty)$. This implies that the all possible concretizations of a node/edge with the linearity property 1 are also in the set of possible concretizations of the same node/edge when we set the linearity property to ω.

Definition 28 (Linearity Property Operations)**.** *Given linearity labels l, l':*

- $\leq$*:* $1 \leq \omega$.
- $l \tilde{\sqcup} l' = max(l, l')$.

Layout. The layout property has a similar construction to the *linearity* domain where each abstract property is defined in terms of a set of concrete properties that *may* hold in the region of the heap that a given node concretizes to. The domain was constructed in a way such that there is a base element *(S)ingleton* that only admits concrete regions that satisfy a single structure predicate and each successive abstract property admits regions satisfying additional predicates up to the *(C)ycle* property which admits concretizations of the node which can satisfy any of the structure predicates.

Definition 29 (Layout Property Operations)**.** *Given layout labels s, s':*

- $\leq$*:* (S)ingleton $\leq$ (L)ist $\leq$ (T)ree $\leq$ (D)ag $\leq$ (C)ycle.
- $s \tilde{\sqcup} s' = \max(s, s')$.

This matches the intuition that if we join a node with the *(S)ingleton* label, which only admits concretizations that satisfy the *Singleton Structure* concrete predicate, and a node with the *(T)ree* label, which admits concretizations that satisfy the *Tree* or *List* or *Singleton*

Structure concrete predicates, we get the *Tree* value which admits any concretization that either of the argument *layout* labels admitted.

Connectivity. First we order the individual *conn* properties $\{$*disjoint*, *connected*, *share*$\}$, which are again defined in such a way that the set of concretizations of a pair of edges and a node that is consistent with the *disjoint* property is a subset of the concretizations consistent with the heap graph which has the *connected* instead, and a similar situation holds for the *connected* and the *share* properties. Thus, we have the total order *disjoint* $\leq$ *connected* $\leq$ *share*. Since the *connectivity* property is a relation on $\hat{E} \times \hat{E} \times \hat{N} \times$ *connR* we need to define the order/join on this relation and we need to consider the particular relations between the edges/nodes in g and g' as given by the subgraph isomorphism $\Pi : g \mapsto g'$.

In order to determine if the connectivity relation *connR* for the abstract heap g is less than the connectivity relation *connR*$'$ of the abstract graph g' we need to ensure that any concrete heap that is consistent with the connectivity relation *connR* is also consistent with the connectivity relation *connR*$'$. In order to accomplish this we need to determine if every entry in the relation $(e_1, e_2, n, c_r) \in$ *connR* has a corresponding entry $(\Pi(e_1), \Pi(e_2), \Pi(n), c'_r) \in$ *connR*$' \wedge c_r \leq c'_r$).

Definition 30 (Connectivity Relation Operations)**.** *Given connectivity relations* connR *for* g, connR$'$ *for* g' *and* $\Pi : g \mapsto g'$*:*

- connR $\leq$ connR$' \Leftrightarrow \forall (e_1, e_2, n, c_r) \in$ connR $(\exists (\Pi(e_1), \Pi(e_2), \Pi(n), c'_r) \in$ connR$'$ $\wedge c_r \leq c'_r)$
- connR$\tilde{\sqcup}$connR$' = \{(e'_1, e'_2, n', \max(c_r, c'_r) \mid (\exists e_1, e_2, n$ *s.t.* $\Pi(e_1) = e'_1 \wedge \Pi(e_2) = e'_2 \wedge \Pi(n) = n' \wedge (e_1, e_2, n, c_r) \in$ connR$) \vee ((e'_1, e'_2, n', c'_r) \in$ connR$')\}$.

Interference. The *interference* property is a unary property and like the *linearity* and *layout* properties we can define a linear order for it based on the definitions, which are

structured to ensure that any concretization of an edge with the *non-interfering* label is also a valid concretization of the edge with the *interfering* (or *aliasing*) label.

Definition 31 (Interference Property Operations)**.** *Given interfere labels* i, i'*:*

- $\leq$*:* non-interfereing $\leq$ interfereing $\leq$ aliasing.
- $i \tilde{\sqcup} i' = \max(i, i')$.

Edge Dominance. Similar to the *connectivity* properties, *edge dominance*, is a relation on the graph. However, unlike the *connectivity* property it is a simple binary relation on $\hat{E} \times \hat{E}$. The reason that we treat this property independently from the *connectivity* relations is that it is a *must* property (it holds for all states) where as the *connectivity* is a *may* property (there may or may not be a state in which the property holds). Thus conceptually, the domain orders are reversed (i.e. the join of the connectivity relation is the *or* of the relations while the join in the dominance will be the *and* of the relations and similarly for the order relation). We want to ensure that if a concrete heap which is consistent with the *edge dominance* properties (the *domEQ* relation) in the abstract heap graph g then the concrete heap is also consistent with the *edge dominance* properties (the *domEQ*$'$ relation) in g', which implies that if a given relation is in *domEQ*$'$ then is must also be in *domEQ* and thus the relation is reversed from the *connectivity* relation.

Definition 32 (Edge Dominance Relation Operations)**.** *Given edge dominance relations* domEQ *for* g*,* domEQ$'$ *for* g' *and* $\Pi : g \mapsto g'$*:*

- $\text{domEQ} \leq \text{domEQ}' \Leftrightarrow \forall (e'_1, e'_2) \in \text{domEQ}'$ $(e'_1, e'_2$ *are not in the range of* $\Pi) \vee (\exists (e_1, e_2) \in \text{domEQ} \wedge \Pi(e_1) = e'_1 \wedge \Pi(e_2) = e'_2)$
- $\text{domEQ} \tilde{\sqcup} \text{domEQ}' = \{(e'_1, e'_2) \mid (e'_1, e'_2$ *are not in the range of* $\Pi) \vee (\exists e_1, e_2$ *s.t.* $\Pi(e_1) = e'_1 \wedge \Pi(e_2) = e'_2 \wedge (e_1, e_2) \in \text{domEQ}) \wedge (e'_1, e'_2) \in \text{domEQ}')\}$.

Node Dominance. A similar situation holds for the *node dominance* property which is also a *must* property. The order definition represents the idea that the only way for the dominance property of n' to admit more concrete states than n is for the set to have fewer restrictions on which reference sets must point to every object in the region (which is equivalent to having fewer dominating edges).

Definition 33 (Node Dominance Operations)**.** *Given node dominance sets* $\{e_1,\ldots,e_k\}$ *for* $n \in g$, $\{e'_1,\ldots,e'_j\}$ *for* $n' \in g'$ *and* $\Pi : g \mapsto g'$*:*

- $\{e_1,\ldots,e_k\} \leq \{e'_1,\ldots,e'_j\} \Leftrightarrow \{e'_1,\ldots,e'_j\} \subseteq \{\Pi(e) \mid e \in \{e_1,\ldots,e_k\}\}$.
- $\{e_1,\ldots,e_k\} \tilde{\sqcup} \{e'_1,\ldots,e'_j\} = \{\Pi(e) \mid e \in \{e_1,\ldots,e_k\}\} \cap \{e'_1,\ldots,e'_j\}$.

Active Iterator/Indexer. The *activeIter* property is much like the *offset* property in the sense that we assume all variable names are incomparable. However, we do allow the join of two *activeIter* properties, $ai \tilde{\sqcup} ai'$ by keeping the *activeIter* property if $ai = ai'$.

Empty, At-First. The empty and at first properties are the instrumentation properties that allow us to track if a given collection is empty and if the current active iterator must be at the start iterator position for the collection.

Definition 34 (Empty Property Operations)**.** *Given empty labels* $\mathrm{ev}, \mathrm{ev}'$*:*

- $\leq$*:* $\mathrm{empty} = \mathrm{true} \leq \mathrm{empty} = \mathrm{unknown} \wedge \mathrm{empty} = \mathrm{false} \leq \mathrm{empty} = \mathrm{unknown}$.
- $\mathrm{ev} \tilde{\sqcup} \mathrm{ev}' =$ *if* $\mathrm{ev} = \mathrm{ev}'$ *then* ev *else* unknown.

Definition 35 (atFirst Property Operations)**.** *Given atFirst labels* $\mathrm{af}, \mathrm{af}'$*:*

- $\leq$*:* $\mathrm{atFirst} = \mathrm{true} \leq \mathrm{atFirst} = \mathrm{false}$ *(they are comparable here since the false value admits all concretizations of the iterator position).*
- $\mathrm{af} \tilde{\sqcup} \mathrm{af}' = \mathrm{af} \wedge \mathrm{af}'$.

7.2 Order and Upper Approximation of Labeled Storage Shape Graphs

In this section we define the order $\leq_g$ relation and a partial function for computing the upper approximation $\tilde{\sqcup}_g$ of two *labeled storage shape graphs*. For the remainder of this section we will assume without loss of generality that we are comparing $g \leq_g g'$ or merging $g \tilde{\sqcup}_g g'$ and the map $\Pi : g \mapsto g'$ is a subgraph isomorphism from g into g'.

Order. Given a method for computing a subgraph isomorphism from $g \mapsto g'$ (if it exists) and the order relations on abstract heap properties we can define the order by first seeing if there is a map from $\Pi : g \mapsto g'$ and if there is then ensuring that under this map all the property labels for the nodes/edges in $\Pi(g)$ are less than or equal to the values in g'.

Given the definition of the labeled storage shape graph from Chapter 3 we can write g as ($\hat{V}$, $\hat{N}$, $\hat{E}$, $\hat{L_n}$, $\hat{L_e}$, *connR*, *domEQ*) where $\hat{L_n} : \hat{N} \mapsto$ (*types*, *linearity*, *layout*, *nodeDom*, *activeIter*, *empty*, *atFirst*) and $\hat{L_e} : \hat{E} \mapsto$ (*offset*, *linearity*, *interfere*), and similarly for g'.

Definition 36 (Labeled Storage Shape Graph Order ($\leq_g$))**.** *Given* labeled storage shape graphs $g \leq g'$ *iff* $\exists$ *subgraph isomorphism* $\Pi : g \mapsto g'$ *s.t.*

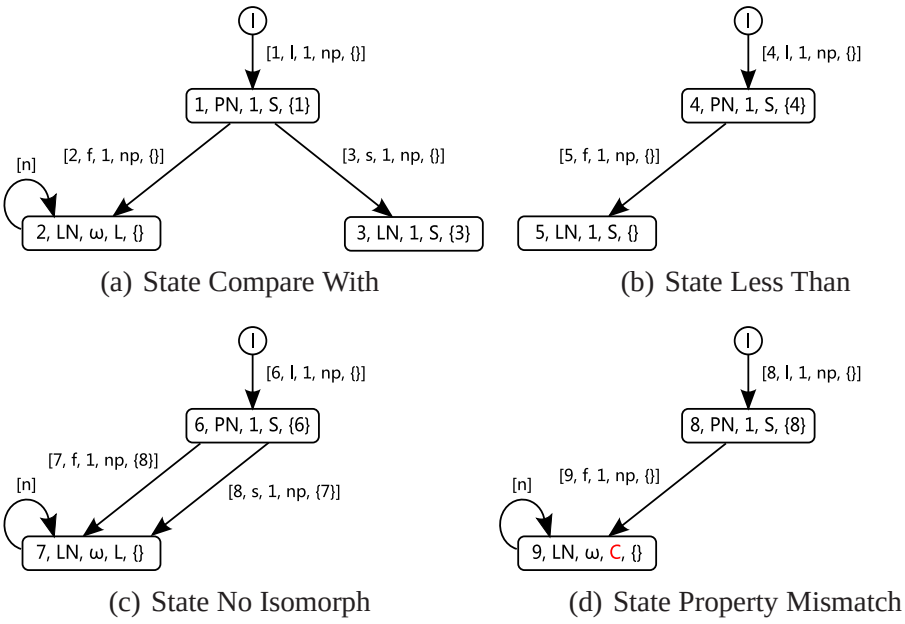
1. $\forall v \in \hat{V}, \Pi(v) \in \hat{V}'$.
2. $\forall n \in \hat{N}, \hat{L_n}(n) \leq \hat{L_n}'(\Pi(n))$.
3. $\forall e \in \hat{E}, \hat{L_e}(e) \leq \hat{L_e}'(\Pi(e))$.
4. connR $\leq$ connR$'$ *under* Π.
5. domEQ $\leq$ domEQ$'$ *under* Π.

In order to ensure the safety of the analysis we must have the property that in the abstract domain $g \leq_g g'$ implies that the set of concrete states that g concretizes to is a

subset of the states that g' concretizes to ($g \leq_g g' \Rightarrow \gamma(g) \subseteq \gamma(g')$). Since there is a copy of g in g', given by Π, we know that any concrete heap that is consistent with the graph connectivity properties of g is also consistent with the graph connectivity properties of g'. We also know that the $\leq_g$ operation ensures that any valid concretizations of a node/edge in g (based on the node/edge property labels) is also a valid concretization of that node/edge (with respect to the property labels) in g' under the map Π. Thus $g \leq_g g' \Rightarrow \gamma(g) \subseteq \gamma(g')$ for the labeled storage shape graphs.

Figure 7.1 contains several simple abstract states that we use to demonstrate the comparison operation. In Figure 7.1(a) we show the model that we compare with the other models. Figure 7.1(b) shows an abstract graph that is less than our comparison model. There is a trivial subgraph isomorphism between the two (even though they do not have identical structures) and all of the instrumentation properties appropriately ordered under the isomorphism. For example in the mapping the node labeled 5 maps to the node labeled 2, the *types* are equal and both the *linearity*, and *layout* properties are consistent with their $\leq$ operations. On the other hand in Figure 7.1(c) we have a case where there is no subgraph isomorphism and so it is not less than the abstract heap in Figure 7.1(a). Figure 7.1(d) shows a graph where there is a subgraph isomorphism but under the mapping node 9 which maps to node 2 are not entirely consistent with the instrumentation properties. In particular the node 9 has the *layout (C)ycle* $\not\leq$ *(L)ist* and thus is not less than or equal.

Upper Approximation. During program analysis with power domains, the possible concrete states of a program are represented as a set of abstract models. In many cases it is possible to have several models in this set that are very similar, differing only in small and often ways that do not affect the precision of the final result (for instance if we have one model that says variable x refers to a *(C)ycle* structure then a model saying x refers to a *(L)ist* structure is redundant as this possibility is already entailed by the first model).



(a) State Compare With (b) State Less Than

(c) State No Isomorph (d) State Property Mismatch

Figure 7.1: Abstract State Comparisons

Carrying these redundant (or only marginally useful) models around is a substantial computational cost. Thus we introduce an upper approximation operator which takes two *similar* abstract heap graphs (we define *similar* to be the existence of a subgraph isomorphism that respects variables and edge *offsets*) and produces a new heap that is an upper approximation of these heaps.

The upper approximation operator ($\tilde{\sqcup}_g$) takes three arguments, graphs g, g' and a subgraph isomorphism $\Pi : g \mapsto g'$. The upper approximation is then computed by computing a pairwise upper approximation of the property labels and relations under the mapping. Thus the result is structurally identical to g' but the labels have been replaced with label values that are larger or equal to the original values.

Definition 37 (Labeled Storage Shape Graph Upper Approximation ($\tilde{\sqcup}_g$))**.** *Given* labeled storage shape graphs g, g' *and a subgraph isomorphism* $\Pi : g \mapsto g'$*:*
$\tilde{\sqcup}(g, g', \Pi) = (\hat{V}', \hat{N}', \hat{E}', \Pi(\hat{L_n}) \tilde{\sqcup} \hat{L_n}', \Pi(\hat{L_e}) \tilde{\sqcup} \hat{L_e}', \Pi(\text{connR}) \tilde{\sqcup} \text{connR}', \Pi(\text{domEQ}) \tilde{\sqcup} \text{domEQ}')$.

This particular join operation allows us to reduce the number of distinct states we must carry in our final power domain while still allowing for (1) important structural differences to be maintained in a fully disjunctive manner and (2) to still use a rich set of instrumentation properties so that if all the heaps in the disjunction satisfy the same property (e.g. the variable x always aliases x and they both point to a list shaped structure) we can precisely model these properties.

To show that the resulting upper approximation $g'' = \tilde{\sqcup}(g, g', \Pi)$ is larger than both g and g' we note that there is a subgraph isomorphism from each of them (Π and the identity) to g'' and by construction each instrumentation property label in g or g' is less than the value that is matched up via the subgraph isomorphism in g''. Thus $g \leq g''$ and $g' \leq g''$

7.3 Full Domain Definition

Given the order operations and the upper approximation operation for the abstract heap graph domain $(\widetilde{H}, \leq_g, \tilde{\sqcup}_g)$ along with the definitions for our domain of equivalent values, $(\widetilde{U})$ Section 4.2, we can construct the complete abstract domain which is used to analyze the programs, $\widetilde{D} = \wp(\widetilde{U} \times \widetilde{H})$. This is a simple power domain over the cross-product of the two base domains. However, to improve the computational performance of the domain we will use a disjunctive join operation and ordering (instead of the usual union and subset operations). We use σ to denote the abstract states in $\widetilde{D}$ and each abstract state σ is a set of models θ each of which is a pair $(\theta.u, \theta.h)$. We use the notation $\theta.u$, $\theta.h$ to access the scalar domain and heap domain components of the tuples.

Definition 38 (Domain Order ($\leq_f$))**.** *Given the order operations defined on the two base domains ($\widetilde{U}$ and $\widetilde{H}$) we define the ordering on $\widetilde{D}$, $\leq_f$ where given two states $\sigma, \sigma' \in \widetilde{D}$:*
$\sigma \leq_f \sigma' \Leftrightarrow \forall(\theta.u, \theta.h) \in \sigma, \exists(\theta'.u, \theta'.h) \in \sigma' \text{ s.t. } \theta.u \leq \theta'.u \wedge \theta.h \leq \theta'.h$

Definition 39 (Incomparable Model Set ($\Delta(\sigma)$))**.** *Given the order operation $\leq_f$ on the models θ we define the set of incomparable values:*
$\Delta(\sigma) = \{\theta \in \sigma \mid \nexists \theta' \in \sigma \setminus \{\theta\}$ *s.t.* $\theta \leq_f \theta'\}$.

Definition 40 (Upper Normal Form ($\Gamma(\sigma)$))**.** *To compute the upper approximation of an abstract state $\sigma = \{\theta_1, \ldots, \theta_k\}$ we apply the following rewrite rule until the set of models is no longer changing:*

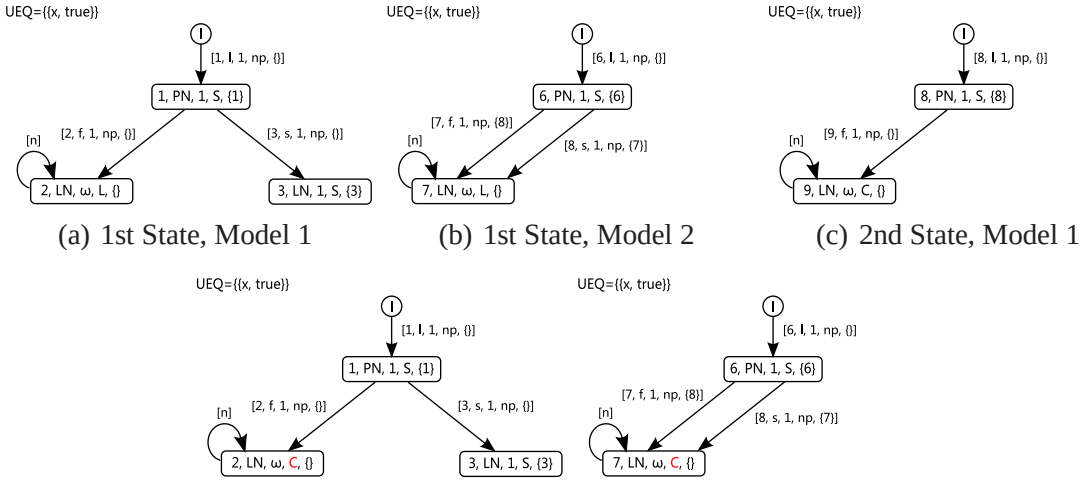
$$\frac{\exists \theta, \theta' \in \sigma \text{ s.t. } (\theta.u = \theta'.u) \wedge (\exists \text{ subgraph isomorphism } \Pi : \theta.h \mapsto \theta'.h)}{\sigma \cup \{(\theta.u, \theta.h \tilde{\sqcup}_g \theta'.h)\}}$$

We define the upper approximation operation $\sigma \tilde{\sqcup}_f \sigma'$ to be the minimal set of $(\theta''.u, \theta''.h)$ such that the $\theta''.u$ values are incomparable or the $\theta''.h$ values cannot be combined via the $\tilde{\sqcup}_g$ operator. This particular definition allows the analysis to differentiate the scalar values in each state precisely, which is critical to modeling integer indexing in arrays and the results of boolean tests, while combining the heap values which is important to minimizing the number of states that are in each state.

Definition 41 (Domain Upper Approximation ($\tilde{\sqcup}_f$))**.** *Given abstract state σ and σ', the upper approximation operator is defined as:*
$\sigma \tilde{\sqcup}_f \sigma' = \Delta(\Gamma(\sigma \cup \sigma'))$.

Example. Figure 7.2 shows the result of performing the disjunctive join operation of one abstract state consisting of two models which are shown in Figures 7.2(a) and 7.2(b) and another abstract state which consists of a single model, shown in Figure 7.2(c). The resulting state consists of the two models shown in Figures 7.2(d) and 7.2(e).

We can see that there are subgraph isomorphisms from the model in Figure 7.2(c) to the two models in Figures 7.2(a) and 7.2(b) however there is no isomorphism between the model in Figures 7.2(a) and the model in Figure 7.2(b). Once we have found the isomorphisms we can use the disjunctive join operation to produce the two models shown in



(a) 1st State, Model 1 (b) 1st State, Model 2 (c) 2nd State, Model 1

(d) Result State, Model 1 (e) Result State, Model 2

Figure 7.2: Disjunctive Join

Figures 7.2(d) and 7.2(e). In our mappings node 2 and node 9 are matched. Since these two nodes have the *layout* properties *(L)ist* and *(C)ycle* respectively the resulting node (2) has the *layout* property *(C)ycle* = $L \tilde{\sqcup} C$ and similarly for the other instrumentation properties in this join and the properties in the second join. The models in Figures 7.2(d) and 7.2(e) are the only 2 incomparable items in the set and also the final state that is produced by the upper approximation.

Concretization (γ_f) and Safety

Definition 42 (Domain Concretization (γ_f))**.** *Given the concretization operations defined for the $\widetilde{U}$ and $\hat{H}$ domains we can define the concretization operator of the full abstract domain:*

$\gamma_f(\sigma) = \{\textit{concrete program state } c \mid c \in \gamma(\theta.u) \wedge c \in \gamma(\theta.h)\}$.

Given the order and concretization operations on the domain we have constructed $\widetilde{D}$ we want to show that they are has the required safety properties with respect to the concrete program states, in particular that $\sigma \leq_f \sigma' \Rightarrow \gamma_f(\sigma) \subseteq \gamma_f(\sigma')$. We

have already shown that for the labeled storage shape graphs $g \leq_g g'$ then $\gamma(g) \subseteq \gamma(g')$. We also know that the relation $u \leq u'$ then $\gamma(u) \subseteq \gamma(u')$ holds in our domain of equivalent values. Given the definition of γ_f above, if $\sigma \leq_f \sigma'$ then $c_r \in \gamma_f(\sigma) = \{\text{concrete program state } c \mid c \in \gamma(\theta.u) \wedge c \in \gamma(\theta.h)\}$ which gives $c_r \in \gamma(\theta'.u) \wedge c \in \gamma(\theta'.h)$ which with the safety properties from the component value and heap models implies $c_r \in \gamma_f(\sigma')$ thus $\gamma_f(\sigma) \subseteq \gamma_f(\sigma')$ as desired.

Chapter 8

Semantics of Primitive Operations

In this chapter we look at the semantics of the primitive language operations in the heap model. In particular we look at the load, store, and pointer comparison operations, which are the backbone of the shape analysis technique. We also look at a number of more specialized, although important, program operations such as type tests and casting, and we look at how array indexing is handled. A key factor in achieving accurate results is the use of a novel technique for partially undoing the *summarization* of information (in Chapter 6 we introduce a bounded representation to summarize unbounded recursive structures and the contents of containers). For efficiency, it is important to make the summary representations as compact as possible. However, this summarization obscures information which is needed to accurately simulate the effect of program statements on the heap model and in order to precisely model the effects of various program operations we perform materialization to make the needed information explicit. While it is possible to perform full materialization (produce an explicit model for each possible configuration represented by the summarized region) on the summarized regions this quickly leads to computational intractability. Thus, the approach presented here is designed to tradeoff some level of accuracy in less common cases to ensure that the worst case, exponential time is avoided and that the method is fast (and accurate) in practice.

8.1 Safety and Precision

In order to ensure that the analysis is sound we must ensure that each of the abstract operations safely approximates the effects of the concrete operation on all of the possible concrete states. Since the MIL language is a subset of Java 1.4 the expression semantics described in this chapter are, except for one minor simplification, the same as described in [23] and more formally defined in [1]. The simplification that we make is to assume that all runtime exceptions (null pointer, array out of bounds, out of memory, etc.) cause immediate program termination.

Definition 43 (Safety of Abstract Semantics)**.** *Given the concretization (γ_f) operator the abstract semantics, $\mathscr{S}[\![f]\!]$, for a concrete program operation f are safe if for all abstract states σ the following holds:*

$$\left\{ f(s_c) \mid s_c \in \gamma_f(\sigma) \right\} \subseteq \gamma_f(\mathscr{S}[\![f]\!]\sigma)$$

That is, the application of the abstract operator results in all the states that can occur in practice and possible more. The relation required is shown as a diagram in Figure 8.1. Since the abstract operations we discuss are all constructive, and use local modifications to the graph structure and enumerative casewise modification to the instrumentation properties, the proofs that these operations are safe approximations of the concrete semantics are simple casewise enumerations of possibilities that follow closely the case analysis in the algorithms. Thus, we limit the discussion to a few non-trivial high-level aspects and otherwise omit proofs.

In general we attempt to define the abstract semantics to be as precise as possible (to minimize the difference between the results of applying an operation in the concrete domain and the abstract domain). However, in many cases in this chapter we heuristically decide, based on our experience with the benchmarks in Chapter 13, to intentionally model an operation in an imprecise manner. Further, we make no general claims about the pre-

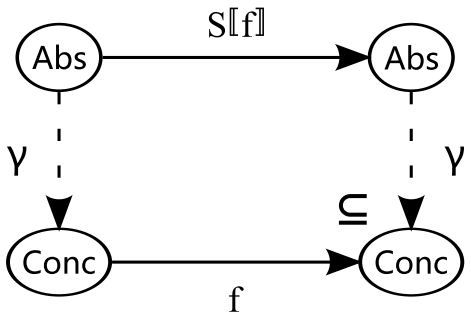


Figure 8.1: Safety Relation Between Concrete and Abstract Operations

cision of the abstract semantics relative to the concrete semantics. Thus, the operations in this section and the associated heuristics for trading precision for performance are topics for further evaluation as we analyze more (and larger programs) and our experience with the analysis increases.

8.2 Materialization

The materialization operation is used to transform single summary nodes into more explicit subgraph representations. In this section we define the materialization operations that (for the most common cases encountered) allow us to undo the summarization by transforming a summary node into a number of nodes (and edges) so that relationships between variables and regions of the heap can be more accurately modeled. The operation is defined for nodes with *(T)ree*, *(L)ist*, or *(S)ingleton* layouts and is further restricted based on the number of incoming edges to the node and the connectivity relations of these edges.

Singleton Materialization. Our materialization operation on *Singleton* nodes is restricted to handle the following cases and otherwise conservatively leave the summary region as it is:

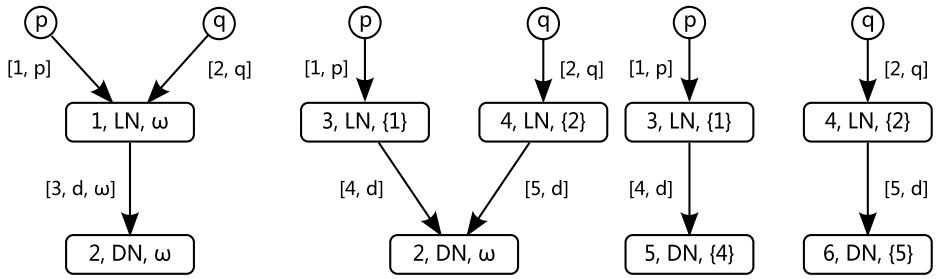
- If the incoming edges can be partitioned into 2 or more equivalence classes based on the *connR* relation.

- If there is a single edge e_c that is *connected* to every other edge and all other edges are pairwise *disjoint*.

The first case is simply when a node represents several disjoint regions of the concrete heap. In this case we use the *refineDisjointEdges* to expand each sub-region into a separate node in the abstract graph. The second is a node where all but one of the edges are disjoint from every other edge. This is handled using the *refineSingleConn* method. In this case we create a new node for each of the edges except the edge e_c, which may be related under the *connR* relation to the other edges, and then splits e_c so that it points to each of the newly created nodes (and thus the possibility that it is connected to any one of the other edges is preserved).

These two operations are a special case of a more general expansion that can be done where we materialize a new node for each possible subset of connected edges based on the *connR* relation. However, in our experience this powerset expansion leads to very small increases in precision at a substantial computational cost (exponential number of cases as opposed to the linear number produced by our heuristics).

In both cases of the singleton materialization we need to split the outgoing edges from the original node into multiple edges (one for each newly created node). This involves two steps, first we create a copy of the edge for each newly created node, and second we need to figure out the *connectivity* relations between these newly created edges. We know that each of these split edges has the same *connected*/*alias* relation with all the other edges as the edge that we are splitting. However the *connectivity* relation with the other newly created split edges depends on the *interfere* property of the original edge. If the original edge was *share* then the split edges *alias*, if the edge was *interfering* then the split edges are *connected* and if the edge was *non-interfering* then the split edges are *disjoint*. We use



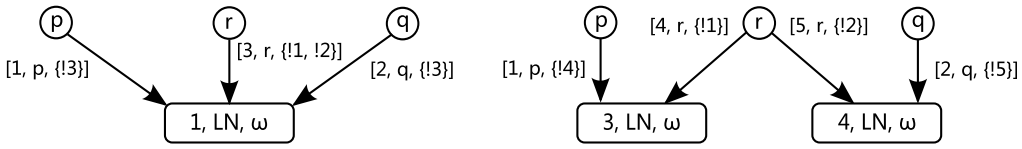
(a) Summarized Singleton (b) Partition Var Edges (c) Partition Pointer Edges

Figure 8.2: Refinement of a region with disjoint sub-regions

the *splitEdge* method to perform this splitting.

Consider the case in Figure 8.2(a) where the variables p, q point to the same node and the edges 1, 2 are not *connected* (the two connectivity lists are empty, the default omitted value). Partitioning results in Figure 8.2(b) where the summary node has been partitioned based on the *connR* relation from the variables. The focus operation has been applied, allowing the model to determine that the split nodes 3, 4 both have the default *linearity* value of 1 and are dominated by edges 1, 2 respectively). The *splitEdge* operation has split edge 3 into two new edges (4, 5). Since edge 3 represents non-interfering pointers (which is the default *interfere* value) the two split edges incident to the node representing the *DN* objects must be *disjoint*. This allows us to apply refinement (and the focus operation) again–the results are shown in Figure 8.2(c).

Figure 8.3 shows a how the *refineSingleConn* method splits a node that has a single shared incoming edge (edge 3). In Figure 8.3(a) we show an abstract heap graph where the two edges (edge 1, 2) do not share but that they both *may* alias with the pointers abstracted by edge 3 (the !3 entries in the connectivity lists). Thus, this model abstracts several concrete heaps, one where all three edges abstract references which do not alias at all, a heap where the references abstracted by edges 1, 3 alias and a heap where the references abstracted by edges 2, 3 alias. However, it excludes a heap where the references abstracted by edges 1, 2 alias (and thus we know that the variables p and q never alias).



(a) Shared Reference (b) Partition pointer Edges

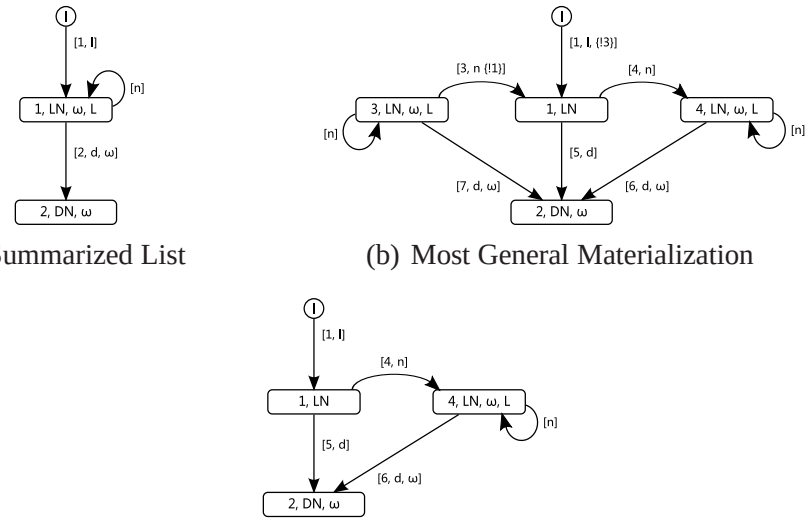
Figure 8.3: Refinement of a region with shared sub-regions

Figure 8.3(b) shows the heap after materialization. We have created two nodes (3, 4) which represent the objects pointed to by edges (1, 2) respectively. We have also created two new edges, edge 4, which represents the possibility that some pointers abstracted by edge 3 alias with the references abstracted by edge 1, and edge 5, which represents the case that some pointers abstracted by edge 3 alias with the references abstracted by edge 2. Thus the materialized figure still represents all the concrete heaps that the original heap abstracted but now the disjoint sets of objects have been made explicit (represented by 2 distinct nodes) instead of implicit (represented by the same node).

List, Tree Materialization. Since our analysis (by construction) will always apply disjoint region materialization before list or tree refinement we know that all the incoming edges to the given list node may be connected. If there are multiple incoming edges to the list or tree we cannot, in general, determine an ordering for them (that is if the edges are *connected* we may not know if they point to the same object or if the target of one is reachable from the other), so we conservatively only consider lists/trees with a single incoming edge and a *linearity* of 1 as candidates for materialization.

In this scenario we make explicit the unique memory location that single incoming reference targets in the list/tree. Since we know that there is a single reference into this data structure it is always the case that this reference targets the head of the list or the root of the tree (any other part of the list/tree is unreachable).

Figure 8.4(a) shows a list with one incoming variable. Figure 8.4(b) shows the most



(a) Summarized List (b) Most General Materialization

(c) Final Materialization

Figure 8.4: Refinement of a node with a list layout

general way in which a list can be referred to by a single program variable; there is a single object that the variable points to, the section of the list that appears before the object the variable points to and a section of the list after this object. We can safely ignore the section of the list before the object that the variable refers to since it is unreachable and therefore cannot affect the program in any way. After we drop this semantically irrelevant portion of the model we have the final result of the materialization, shown in Figure 8.4(c).

Note that in Figure 8.4(c) we have split the edge that summarized the `(d)ata` pointers of all the elements in the linked list into two new edges, one (edge 5) representing the single `(d)ata` pointer that is stored in the head element of the list and an edge (edge 6) representing all the pointers stored in the tail of the list. We perform the materialization of the `DN` node lazily waiting until we need to normalize the graph or until we encounter a load operation on the `(d)ata` field.

The tree materialization operation is nearly identical to the list materialization except that for a k-ary tree we must create k summary sub-trees and a single parent node (instead of a single summary tail and the single head node). Thus, we omit further description.

Full Materialization For Loads. Algorithm 7 shows the code for the materialization operation that fully materializes a target node when we are performing the load operations (before we do the load we want to fully materialize the node that is the target of the load). In this algorithm we first try to split all the disjoint partitions. Then we attempt to apply the second *Singleton* splitting rule to the partition. Finally, if possible, we apply the *List*/*Tree* refinement operations.

Algorithm 7: Refine Load

```
input : graph g, node n
E_p ← partition of the incoming edges to n;
if E_p has 2 or more partitions then
    g.refineDisjointEdges(n, E_p);
    for each newly created node n' do
        g.focus(n');
allcp ← ∃e' s.t. all edges connected or alias to e';
owdisjoint ← all edges except e' are pairwise disjoint;
if allcp ∧ owdisjoint then
    g.refineSingleConn(n, e');
    for each newly created node n' do
        g.focus(n');
if n.layout List or Tree and single in edge of size 1 then
    g.refineListTree(n);
    for each newly created node n' do
        g.focus(n');
```

8.3 Assign, Load and Store

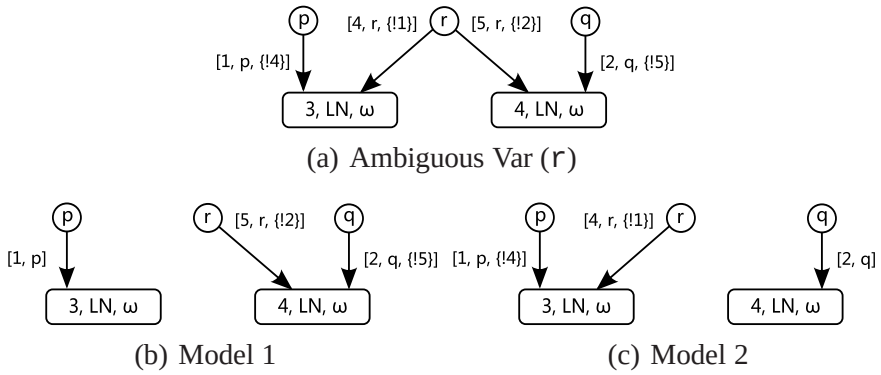
Variable Assignment. The variable assignment operation (`x = y`) does not need to perform any complex manipulations of the heap and can simply clear all the targets of `x` and for each target of `y` create an edge that points to the same node, has the *alias* sharing

property and is *domEQ* to the edge that represents the target of `y`.

Load. The load operation (`x = y.f`) is more interesting as we may need to deal with ambiguous targets of `y.f` (there may be multiple targets of `y` and each of these targets may have multiple outgoing edges with the label `f`) and we may need to materialize the summary node, referred to by `y.f`, into a more explicit representation, Algorithm 7, before we actually update the target of the variable `x`. The other potential complication is self edges.

To get maximal precision we resolve the problem of ambiguous edges by creating a new copy of the model for each possible target and assuming that each of the other edges must have an empty concretization (or in the case of array loads ignoring them). Although this results in potentially exponential growth in the number of models we have found that in practice the problem is minimal (in combination with our disjunctive domain the number of models is usually between 1-4). Our experience indicates that small increase in runtime is much less problematic than the precision that is lost by removing the ambiguity by simply merging all of the edge targets, which ensures that no additional models are created in the load but that often also merges very distinct parts of the heap, resulting in severe losses of information. In Algorithm 8 we assume that the base variable (`y`) and the target of the load (field `f`) have been uniqueified as needed (either by creating multiple models or by merging edges/nodes).

Figure 8.5 shows an example of a model where we may need to resolve the ambiguous edges when performing a load operation. In particular assume we want to simulate the effects of the statement `r.data == null` on the abstract model shown in Figure 8.5(a). Since we do not have a unique target for the variable `r` (it has two outgoing edges 4, 5) we need to resolve the ambiguity by creating two new models (one for each edge). This results in two new models shown in Figures 8.5(b) and 8.5(c) which together represent the same set of states as the model in Figure 8.5(a) but now the target of `r` is unique allowing



(a) Ambiguous Var (`r`)

(b) Model 1

(c) Model 2

Figure 8.5: Resolution of Ambiguous Target Edges

for simpler (and potentially more precise) simulation of the load and comparison.

This is particularly important when we come to the store operation which can be precisely simulated if the store is into a unique location (the model is *strongly updatable*). In this case we can remove the edge representing the old value stored in this location and replace it with the new value, otherwise we must conservatively assume the either value could be stored in any of the (non-unique) locations, which can have a major impact on the precision of the analysis.

Definition 44 (Strongly Updateable)**.** *A memory location represented by a node n is strongly updatable if n.*linearity $= 1$.

This is based on the observation that if we write to a memory location represented by a node of *linearity* 1 through a variable `x`, in a statement such as `x.f = y` then there are two possibilities. One is that `x` refers to the unique object and thus the field `f` must have its old value overwritten, and thus we know the location is updated. Or alternatively that `x` does not refer to the unique object and thus must be null. However, this implies that a null pointer exception will be raised and the program terminates, thus we may safely assume any properties hold after the statement. In either case the removal of the old value and replacement with the new one is safe.

Algorithm 8: Load External, Unique Edge (`x = y.f`)

input : graph *g*, var *x*, var *y*, offset *f*

nullify *x*;

if *y.f is non-null value* **then**
  nullify *x*;
  g.materializeAndLoad(the unique target of *y.f*);
  e ← the unique edge at *y.f*;
  assign *x* to refer to the target of *y*, and mark as *alias* with *e*;
  if *the storage location of y.f is strongly updateable* **then**
    set *x* dominance equal to *e*;
    if *e dominates the target node* **then** set *x* dominates target node;

For a load (`x = y.f`) on a self edge we need to update the internal connectivity relationship (*connR*) to reflect the new connectivity relations between `x` and `y`. Similar to the what is done for the connectivity properties in [20], we update the connectivity relation of `x` to be identical to *may* share with any edge `y` is related with. We then add the connectivity information for `x` and `y` keeping in mind the fact that if the layout is non-cyclic then `x` and `y` cannot alias.

Figure 8.6 shows the interplay between the load and the materialization operations. In Figure 8.6(a) we start with a linked list (the node with the self edge has a *(L)ist* layout) where each `LN` node contains a pointer to a unique `DN` object (edge 3 has all *non-interfering* pointers the default omitted value).

Figure 8.6(b) shows the heap that results from loading off the `l.n` field. In this figure we have split the summary list node (node 2) into a single node representing the exact target of the pointer (node 3) and the remaining tail of the list (node 4). The operation has also split the summary `(d)ata` edge (edge 3) into an edge representing the single pointer stored at the `(d)ata` field of the object abstracted by node 3 and the set of pointers stored at the `(d)ata` fields in the objects stored in the tail of the list (edge 6). Since the original

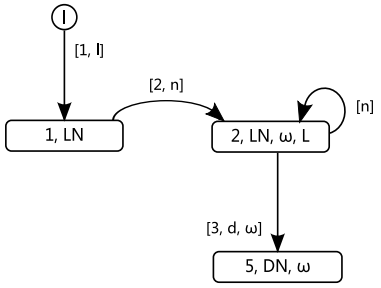
edge that was split (edge 3) contained all *non-interfering* pointers we know that the new edges must be *disjoint*. We also note that since we know that the target of variable `x` is the same object as the pointer stored at `l.n` we marked the edges 2, 7 as *domEQ*.

In Figure 8.6 we have the result of loading from `x.d`. To simulate the effects of this operation the analysis has split node 5 into two new nodes (via *singleton materialization*), node 6 which represents the single object pointed to by `x.d` and node 7 which represents all the `DN` objects stored in the tail of the list. Since we know that edge 5 and edge 6 are *disjoint* the analysis knows that these sets of objects must be disjoint as well and can create two distinct nodes to represent the sets of objects. Since edge 5 represents at most 1 pointer (the *linearity* is the default value 1) and node 6 has no other incoming references we know it must represent at most one object as well (and thus has the default omitted *linearity* value of 1). Finally we note that since we know that the target of variable `y` is the same object as the pointer stored at `x.d` we marked the edges 5, 9 as *domEQ*.

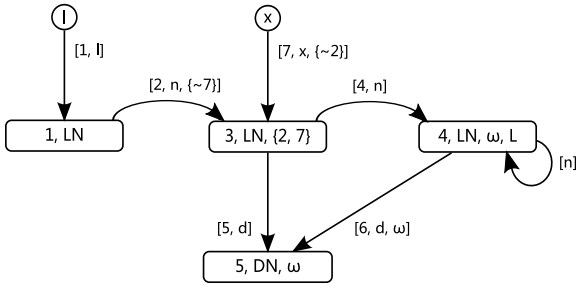
Store. The store operation (`x.f = y`) begins by ensuring that there is a unique target node that `x` refers to (using the same techniques as for the load operation). The analysis then determines if the location at `x.f` can be strongly updated or results in an internal store (`x` and `y` refer to the same node).

If the node (*n*) is strongly updateable then we can remove the edge stored in the target offset. Once we have completed the testing and removal of any edges stored at the field `f` we create a new edge for each possible target of `y`. Just as in the case of the loads these new edges are each dominance equal to the edge representing the target of `y` and if the edge representing the target of `y` dominates a node then so does the newly created edge.

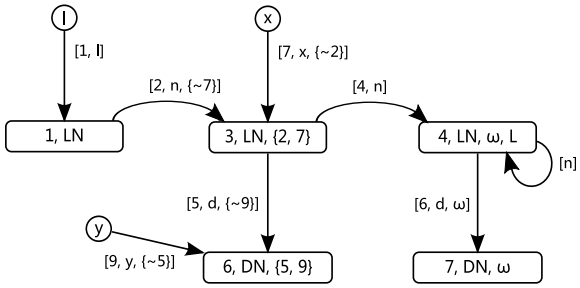
If the node is not strongly updatable we can simply add a new edge to represent the newly created pointer. However, we also have to consider the effect of the store on the dominance information. In particular we cannot accurately determine if a pointer value that was critical to one of the dominance properties was overwritten by this store. To



(a) Initial List



(b) List Load (`x = l.n`)



(c) Data Load (`y = x.d`)

Figure 8.6: Loads on With Materialization, Recursive and Summary Regions

handle this safely we must remove any edges that start at this node (with the same offset as the store is at) and remove them from any node or edge dominance relations that they are in.

We have a similar situation for the internal stores as we did for the internal loads. We update the internal connectivity relation for any variables connected to `x` and `y` to now be connected to each other (but unless they may have aliased before they cannot

alias after the store). We must also update the *interfere* property of any incoming edge which is connected to both `x` and `y` to be *interfere* because this store may have created a path between two of the references abstracted by the given incoming edge. Finally we conservatively update the shape of the node to *(C)ycle*.

8.4 Reference Variable Comparison

When performing equality tests we generate one version of the abstract heap for each possible outcome. For a nullity test of a variable we create one model in which the variable *must* be *null* and one model in which the variable is *non-null*. In the case where the variable is assumed to be *null* we are asserting that the concretization of the edge that represents the variable target is empty. Thus, if the variable dominates a node we infer that the node does not represent any live objects and all the other incoming/outgoing edges must also have empty concretizations. Similarly any edge that is *domEQ* to the variable must also have an empty concretization (and can be removed from the graph). Algorithm 9 gives the code for the case where we assume the variable *must* be *null*.

Algorithm 9: Assume Var Null (`v == null` is *true*)

input : graph g, var v

$E_s \leftarrow$ all edges that represent the targets of v;

$E_{\text{null}} \leftarrow \emptyset$;

for *edge* $e \in E_S$ **do**

  $E_{\text{null}} \leftarrow E_{\text{null}} \cup \{e' | e' \ domEQ \ e\}$;

  $n \leftarrow$ the target node of e;

  if e dominates n **then** $E_{\text{null}} \leftarrow E_{\text{null}} \cup \{$all incoming edges to $n\}$;

for *edge* $e \in E_{null}$ **do**

  g.removeEdge(e);

In the case of an equality comparison between two variables which *may* be *non-null*

`x == y` we can strengthen the information we have in the models that represent the true and false branches. For both the *true* and *false* cases we begin by ensuring that `x` and `y` have unique targets. In the case where we assume this test returns true, if `x` and `y` refer to different nodes in the graph then the only way they can be equal is if both variables are `null`. If not, we add the fact that `x` *domEQ* `y` to the model. In the case where we assume this test must be false then we can check if the relation `x` *domEQ* `y` holds and if it does we can rule this path out as being infeasible. Also as special case, we know that if `x` dominates the node and in the connectivity relation `x` and `y` cannot alias then `y` must be null (otherwise `y` refers to an object in the region abstracted by the node and since `x` dominates this region it also must refer to this object but the model tells us that they do not alias, which is infeasible). Thus,we can safely assume that `y` is *null*.

Nullity Example With Dominance. Figure 8.7 shows an example of how the dominance information plays an important role in accurately simulating the effects of the nullity tests. In Figure 8.7(a) we show the abstract heap that represents the state of the program during a loop that is iteratively traversing the list pointed to by `l` and nullifying each of the `(d)ata` fields. We see the single `LN` node referred to by `l` a *(L)ist* shaped section of the heap of unknown length (node 3), all of which have had the `(d)ata` field nullified, followed by node 5 which represents the single object `x` points to and then the tail of the list. These last two nodes represent the objects in the list that have not been processed and still may have *non-null* fields. Note that during the list traversal the load operation `x = x.next` gives the dominance equality relation between the edge that represents the reference from `x` (edge 7) and the edge representing the pointer from the previous list node (edge 3).

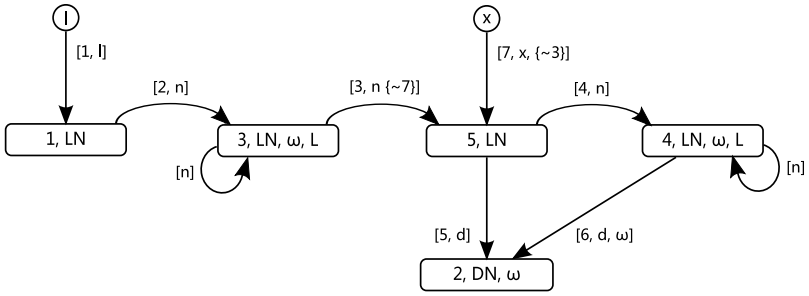
When we encounter the exit test for the loop `x == null` we produce one model in which the condition is assumed to be false and the loop is executed again and one model in which the condition is assumed to be true and the loop is exited. Figures 8.7(b) and 8.7(c)

show the result of assuming that the condition is true (i.e. `x` *must* be `null`) without and with the use of the dominance information. Figure 8.7(b) shows the heap that results when the dominance information present in the model is not used (e.g. we simply remove all the edges that represent possible reference targets of the variable `x`). In this case we have simply removed edge 7 from the model and the analysis has (overly) conservatively assumed that there may be entries in the list with *non-null* pointers. However, the dominance information captures the conditional dependence between the pointer value of the previous load and the current target of the variable `x` by encoding that they have identical reference targets. Thus the analysis can use this dominance equality to infer that if `x` is *null* then the previously loaded pointer is also *null*. The result is shown in Figure 8.7(c) where the analysis has removed edge 3 as well as edge 7 and then since the tail of the list is unreachable it has been removed as well. Thus the analysis can correctly identify that all of the elements in the list have had the `(d)ata` field nullified.

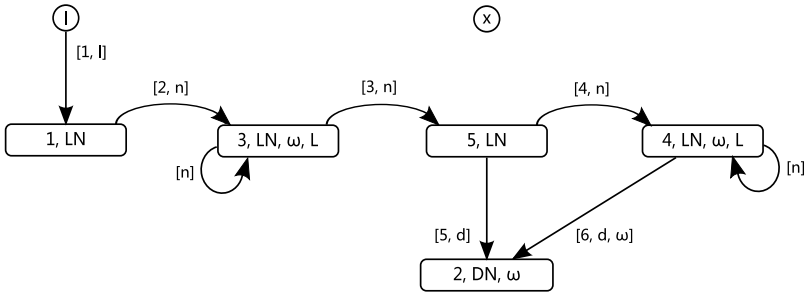
8.5 Array Operations

Array Load/Store. When dealing with arrays we can use the same approach as for the load/store of fields in objects but we must also deal with the complication that each array may be partitioned into multiple summary regions by an indexing variable, the pointers stored in indecies that are less that the index variable the pointer stored at the index given by the variable and the pointers stored in the indecies greater than the index variable. The variable that this partitioning is done on in may either the same variable the load is being performed on or a different one. For the load on an array we first define a method that initializes a new partition based on a given indexing variable (i.e. it forgets the old partition and repartitions on the new index variable).

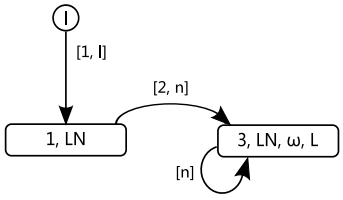
Algorithm 10 takes an array and forgets the current indexing variable partition. This is done by replacing the special partition offsets (*ai*, *at*, *bi*) in any edges with the generic



(a) Initial State



(b) No Dom Use



(c) With Dom Use

Figure 8.7: Equality Test With Null (`x == null` is *true*)

position offset (*?*). After this replacement has been done we re-normalize the array by removing any *equivalent* edges (this in effect merges each set of three partitioned edges *ai*, *at*, *bi* into a single edge *?*).

Once we have forgotten the old indexing information we can the split the (potentially multiple) summary edges based on their order relation to the new indexing variable. This assumes the node representing the array has *linearity* 1, if not we conservatively do not

Algorithm 10: clearIndexer

input : g a heap graph, n a node

if *n has an active iterator* **then**
    n.activeIter $\leftarrow \perp$;

    foreach *out edge e* **do**
        if $e.offset \in \{\text{ai}, \text{at}, \text{bi}\}$ **then** $e.offset \leftarrow ?$;
g.removeEquivalentEdges(n);

split any edges, and simply load using Algorithm 8, replacing the field `f` with the offset *?* and references to `y.f` with `y[j]`. Thus for the remainder of this section we assume the target node has *linearity* 1.

Since an array may have multiple outgoing summary edges (edges with the offset *?*, see Section 6.2) each of which represent pointers stored in the array, we must ensure that the load operation covers the possibility that index `j` could refer to a pointer in each of these edges. Thus we create a new model for each summary edge and split all the summary edges into partitioned edges accordingly (i.e. we create a new model for each out edge and assume that index `j` refers to a pointer represented by that edge).

For the i^{th} splitting we assume that the i^{th} summary edge contains the pointer that will be loaded. For this edge we split it into three new edges, the single element that the index variable refers to, all the elements that come before the index location and the set of elements that come after the index location. For the rest of the summary edges we simply split them into two new edges, one representing all elements that come before the index location and one representing all the elements that come after the index location. The edge splitting is done via the *splitEdge* method from Section 8.2

We have a special case that we want to handle in this operation when $j = 0$, the load is from the first index in the array. In this case we set the *first* flag to *yes* and we do not create split out any edges with the offset `bi` since we know there are no indecies that are less than the current index variable (`j`).

Algorithm 11: splitIthSummary

input : *m* model, *n* node, *j* index var, *i* edge to split, *first* flag (yes if $j = 0$)

$e_{ith} \leftarrow$ ith summary edge;

if *first = yes* **then**
    in *m* split e_{ith} into 2 edges, *at*, *ai*;

else
    in *m* split e_{ith} into 3 edges, *bi*, *at*, *ai*;

set *at* edge to *linearity* 1;

for *all other summary edges* e_s **do**

    if *first = yes* **then**
        in *m* transform e_s from a *?* summary edge to an *ai* edge;

    else
        in *m* split e_s into 2 edges, *bi*, *ai*;

n.activeIndexer = *j*;

Once we have the algorithms to remove and introduce index variable partitions into the arrays we can define the array load operation, again we assume that the array target (given by variable A) is a node of *linearity* 1, if it is not we just use the base load algorithm on the *?* offset). The pseudo-code to do this load is shown in Algorithm 12. In this algorithm we first make sure the node that represents the array we are loading from is partitioned on the indexing variable we want to use. If the indexing variable is already correct we continue on otherwise we forget the current indexing variable and for each summary edge we create a new model assuming that the element we are going to load is in that summary set of pointers. Once we have done this (using the *splitIthSummary* method) we can simulate the load operation much as we did in the field load (only now we use the special field name *at* to identify the edge that represents the pointer we are loading).

Figure 8.8 shows the array load algorithm (`y = A[j]`) applied to an array with two summary edges. The array in Figure 8.8(a) has 2 summary edges (one containing non-recursive DN objects and the other containing recursive *LN* objects). Figures 8.8(b) and 8.8(c) show the two heap models created by the load. In the first one, Figure 8.8(b) we

Algorithm 12: Load Array (`x = A[j]`)

input : model *m*, target *x*, array *A*, index *j*

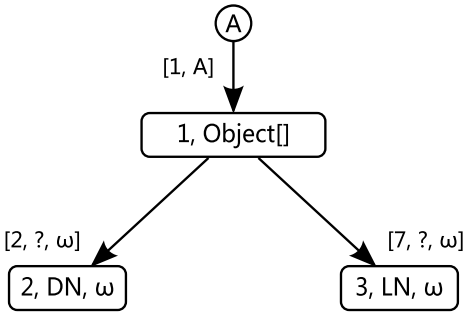
output: model[] r

$n \leftarrow m$.target(A);

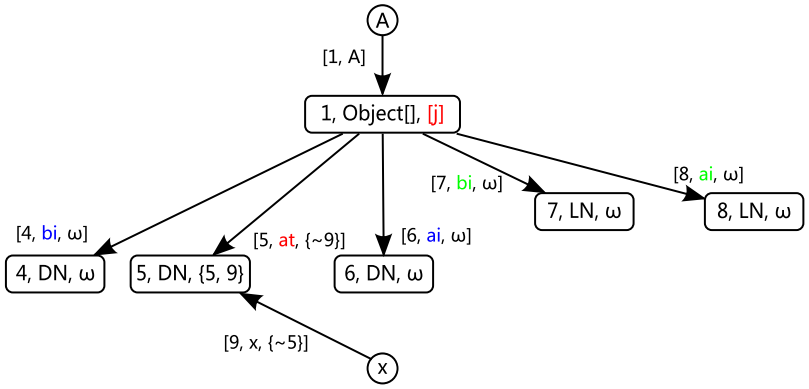
if *n.activeIndexer = j* **then**
    $r \leftarrow$ new model[1];
    $r[0] \leftarrow m$;
else
    n.clearIndexer();
    $r \leftarrow$ new model[*n*.summaryCount()];
    $f \leftarrow m.mustEqual(0, j)$ *? yes : no*;
    for $k \in 0 \ldots r.\text{Length} - 1$ **do**
        r[k] $\leftarrow$ *m*.copy().splitIthSummary(*k*, *r*[k].target(*A*), *j*, *f*);
for $i \in 0 \ldots r.\text{Length} - 1$ **do**
    $m' \leftarrow r$[i];
    m'.materializeAndLoad(the unique target of $A[j]$);
    $e \leftarrow$ the unique edge out of *n* with the offset *at*;
    assign *x* to refer to the target of, and mark as *alias* with *e*;
    if *the storage location of* $A[j]$ *(the* at *edge) is strongly updateable* **then**
        set *x* dominance equal to *e*;
        if *e dominates the target node* **then** set *x* dominates target node;

assume that the variable `j` refers to an array index that contains a reference to one of the `DN` objects and have loaded and performed materialization on the target. Figure 8.8(c) shows the other possibility where we assume that the variable `j` refers to an array index that contains a reference to one of the `LN` objects.

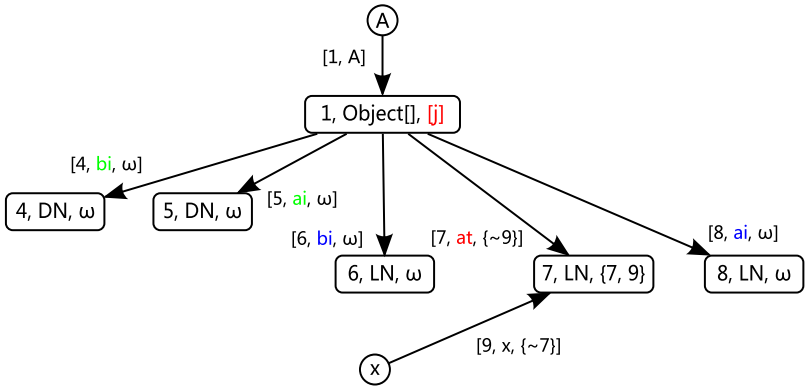
Index Var Increment and Test. The abstract semantics of the variable increment operation and the order tests need to be updated to account for the possibility that the variable is being used to index through an array or a sequence based collection (`ArrayList`,



(a) Multiple Summary Edges



(b) Load Model 1



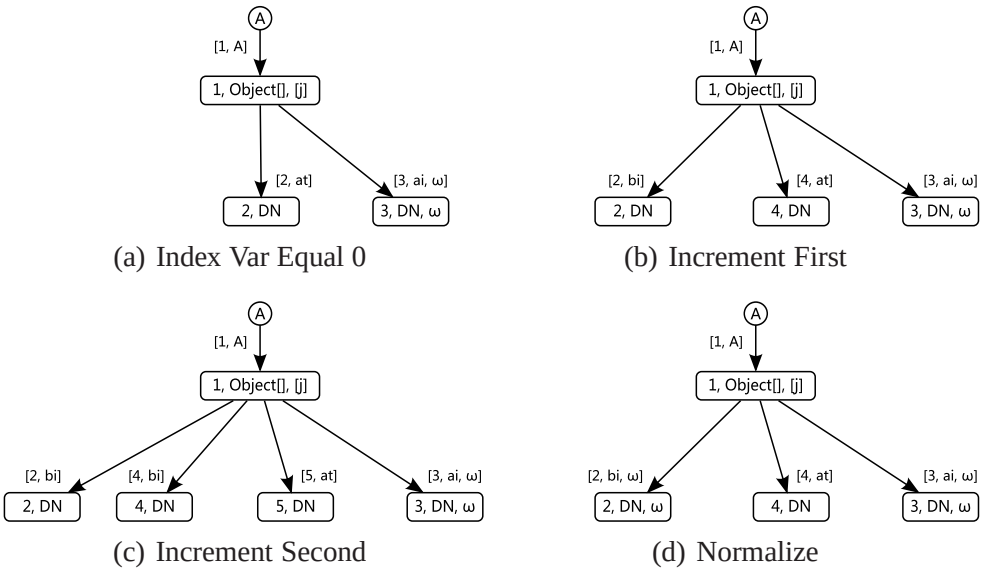
(c) Load Model 2

Figure 8.8: Load Array (`x = A[j]`)

`Vector`, etc.). Thus we must provide abstract semantics for the increment operator (`i++`) and the common case of comparing against an array length (`i < A.Length`). While there are many other possible forms of indexing and comparison that can be used in a program these two are by far the most common that appear and the semantics for the other constructs can be computed in a similar way.

We also note that there is an important difference in how we update the partitions of the summary regions in the array from the approach taken in [22]. While our approach uses the length test to determine when the end of the array is reached, updating the state of the model and separating states where `i < A.Length` is true from the states where it is false, while the approach in [22] creates additional models during the increment step (one where there are many elements left in the array, one where there is exactly one element left in the array and one where there are no elements left in the array) and then filters them at the test. This particular approach makes it easy to model algorithms that treat the last element in the array differently (while our approach requires some additional implementation effort) but it also imposes a non-trivial computational cost for the most common case of simple incrementing from 0 to the length of the array. This difference is fundamentally based on our approach of lazily generating states whenever possible instead of an eager state generation approach.

The var increment operation (`i++`) is structured similarly to the first part of the array load algorithm. To simulate the semantics of the array index operation on a collection that has a number of summary after edges (edges with the offset `ai`, which represent pointers stored in indecies larger than the current index) we need to assume that the increment operation could cause the current index to refer to a pointer in any of these edges. We use the method *afterCount* which returns the number of edges with the `ai` label to compute the number of new models that need to be generated to cover all possibilities. The algorithm first sets the current index location to before the current index location (*at* edge goes to *bi*) and then splits the i^{th} after index edge (*ai*) into a new current index location edge (*at*) and



(a) Index Var Equal 0 (b) Increment First

(c) Increment Second (d) Normalize

Figure 8.9: Increment of an Array Index Variable

the remaining index locations *ai*. This is accomplished using the *splitIthAfterIndex* which is defined in the same was as the *splitIthSummary* method.

Algorithm 13: incrementIndex

input : model *m*, array var *A* (refers to collection of linearity 1), index var *j*

output: model[] r

m.forgetScalarValue(*j*);

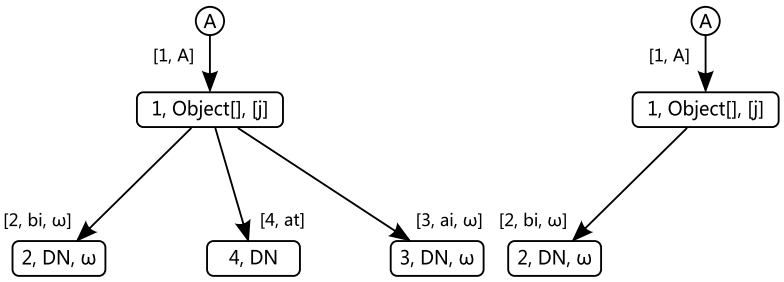
$n \leftarrow m$.target(*A*);

$r \leftarrow$ new model[*n*.afterCount()];

foreach *out edge e in n* **do**
  if *e.offset* = at **then** *e.offset* $\leftarrow$ *bi*;

for $k \in 0 \ldots r$.Length $- 1$ **do**
  $m' \leftarrow m$.copy();
  $n' \leftarrow m'$.target(*A*);
  r[k] $\leftarrow m'$.splitIthAfterIndex(k, n', j);

Figure 8.9 shows the array increment algorithm applied to an array with one summary



(a) Base Array (b) `j < A.Length` is *false*

Figure 8.10: Compare Index Var with Length (*false* model)

after edge. The array in Figure 8.9(a) has a single summary partition that is being indexed by the variable `j`. In this particular figure shows the state when $j == 0$, thus there are only the `at` edge representing the single object that `j` refers to (is at `A[0]`) and the (*ai*) edge representing all the elements after the location `j` refers to (the entries in indecies $1 \ldots A.Length - 1$). Figure 8.9(b) shows the model that results from incrementing `j` (`j++`). In this figure we have renamed the old `at` edge to `bi` since it represent the entry at index 0 and now $j = 1$ and we have split out a new edge from the `ai` edge to represent the new entry `j` refers to. Figure 8.9(c) shows the heap after a second increment operation (`j++`) where we now have 2 `bi` edges representing the two entries that come before the current position referred to by `j` in the array. Finally, in Figure 8.9(d) we show the result of *normalizing* the heap where we have joined the two *equivalent* edges (edges 2 and 4) into a new summary edge. This figure shows the most general state when processing the array incrementally, when we have some unknown number of entries that come before `j` (`bi`) the single entry that `j` refers to (`at`) and some number of entries that come after `j` (`ai`).

The comparison operation for indexing variables and array lengths `i < A.Length` performs the lazy splitting of the model into a model where $i \geq A.Length$ is true and a model where $i < A.Length$ is true. If $i \geq A.Length$ is true then the current element referred to in the array `A` is non-existent and the set of entries that come after the index `i` in the

array `A` is empty. Thus we can remove the edges with the *offsets* `at` and `ai` and any nodes they *dominate* along with any edges that they are *dominance equal* with (just like in the pointer comparison operations).

Figure 8.10 shows the comparison algorithm (`j < A.Length`) applied to the array in Figure 8.10(a) which represents the state of our abstract heap model after many passes through a loop where `j` is incremented. In this model the variable Figure 8.10(b) shows the heap model created for the *false* possibility (the *true* possibility leaves the model unchanged). This figure shows how the analysis uses the fact that the index variable now points off the end of the array to update the model to remove the possibility where `j` refers to and entry in the array or there are any entries in later indecies in the array, the `at` and `ai` edges respectively.

8.6 InstanceOf and Cast

In this section we look at two operations that are commonly used in object-oriented programs for determining the types of objects at runtime. Precisely modeling these operations is critical to modeling many programs. In particular in many Java programs the lack of template collections prior to the Java 1.5 specification results in many object casts when retrieving elements from collections.

To simulate the semantics of the `instanceof` operation in the abstract domain we again create two models to represent the true and false results of the test. The first model represents the state of the heap if the test returns true. In this case we know that the object pointed to must be of a subtype of the type specified by the instance of test. If the node pointed to by the test variable is strongly updateable, Section 8.3, we can set the *type* property to the intersection of the original *type* set and the set of all subtypes of the type given in the test. Otherwise we must conservatively assume that the *type* property is

unchanged (since the node represents many objects the other objects in the region could be of any type). The second model represents the case where the object is of any type other than a subtype of the type given in the test. The algorithm for the full `instanceof` test is shown in Figure 14.

Algorithm 14: instanceof

input : model m, var x type, t

output: model[] r

$r \leftarrow$ new model[2];

$t_s \leftarrow$ all subtypes of t;

$m_t \leftarrow m$.copy();

$r[0] \leftarrow m_t$;

$n_t \leftarrow m_t$.target(x);

if n_t.*linearity* = *1* **then**
    n_t.types $\leftarrow n_t$.types $\cap\ t_s$;
    focus(n_t);

$m_f \leftarrow m$.copy();

$r[1] \leftarrow m_f$;

$n_f \leftarrow m_f$.target(x);

if n_t.*linearity* = *1* **then**
    n_f.types $\leftarrow n_f$.types $\setminus\ t_s$;
    focus(n_f);

The abstract semantics for the cast operation are similar in structure to the instance of operation. However for the cast operation we can assume that the cast always succeeds (if it does not we have an exception which we handle separately). Thus the cast operation only produces a single model that is identical to the true model in the instanceof test.

Chapter 9

Dataflow Analysis

In this chapter we examine how the dataflow analysis of a MIL program is done. The general approach we take is to use as much structural information from the program as possible to improve the performance of the analysis. Thus we split the analysis into two components, a local phase, which uses a fairly standard structured dataflow analysis, and an interprocedural component which uses a hybrid exploration approach to traverse the call graph and employs a novel method to manage the state explosion issues that arise when performing a context-sensitive program analysis.

9.1 Local Flow

In Chapter 8 we defined the primitive semantics for the basic program operations (assign, load, store, compare, type test, etc.). Given an abstract state σ (as defined in Section 7.3) and an expression *e* we use the notation $\mathscr{S}[\![e]\!]\sigma$ to denote the semantics of the expression in the abstract state σ and similarly for statement *s*, the abstract semantics are denoted by $\mathscr{S}[\![s]\!]\sigma$. In this section we show how to compose the semantics of these primitive operations using the control flow primitives in the language (*if-else* and *while*).

Abstract Conditional Semantics. Using the equivalent values information (from Section 4.2) and a boolean condition b (a literal or a variable), we can filter an abstract state $\sigma = \{\theta_1, \ldots, \theta_k\}$ into two new abstract states based on the possible truth of b.

Definition 45 (Partition σ based on truth value of b)**.** *Given an abstract state* σ *and a boolean variable* b *define:*

$$\sigma_{\text{true}} = \mathscr{S}[\![b]\!]_{true}(\sigma) = \{\, \theta \in \sigma \mid b \text{ may } \textit{be true in } \theta.u\}$$
$$\sigma_{\text{false}} = \mathscr{S}[\![b]\!]_{false}(\sigma) = \{\, \theta \in \sigma \mid b \text{ may } \textit{be false in } \theta.u\}$$

Using the above definitions we can write the standard definition for the `if` statement, $\mathscr{S}[\![if(b)\ block_t\ else\ block_f]\!]\sigma = \mathscr{S}[\![block_t]\!](\sigma_{true}) \cup \mathscr{S}[\![block_f]\!](\sigma_{false})$. However, using this definition of the semantics can result in exponential growth in the number of states that the analysis must deal with (since for most cases at the union of the abstract states that result from analyzing *true* and *false* branches will have many models that are nearly identical).

To avoid this we use the upper approximation operator $\tilde{\sqcup}$ instead of a simple set union at the control flow join. This allows the analysis work with a substantially smaller set of models at a small loss of precision, as shown in Figure 7.2 in Section 7.3.

Definition 46 (Semantics of `if-else`)**.** *Given the conditional control flow statement* `if`(b) $block_t$ `else` $block_f$ *the abstract semantics are:*

$$\mathscr{S}[\![if(\text{b})\ \text{block}_t\ else\ \text{block}_f]\!]\sigma = \mathscr{S}[\![\text{block}_t]\!](\sigma_{\text{true}}) \tilde{\sqcup} \mathscr{S}[\![\text{block}_f]\!](\sigma_{\text{false}})$$

Abstract Loop Semantics. The semantics of a looping statement `while` can be expressed in terms of accumulating all possible exit states.

Definition 47 (Semantics of i^{th} loop iteration)**.** *Given a loop statement,* `while`(b) block, *the abstract state of the heap at the loop test for the* i^{th} *iteration of the loop is:*

$$\sigma_i = \begin{cases} \sigma & \text{if } i = 0 \\ \mathscr{S}[\![\text{block}]\!](\mathscr{S}[\![b]\!]_{true}(\sigma_{i-1})) & \text{otherwise} \end{cases}$$

Then we can define the semantics of the loop analysis as the upper approximation (Section 7.3) of all the possible exits from the loop.

Definition 48 (Semantics loop statement (`while`))**.** *Given the loop* `while`(b) block*, and the* upper normal form *operator,* Γ*, and* incomparable model set *operator,* Δ*, from Section 7.3, the abstract state of the heap at the loop exit is:*

$$\mathscr{S}[\![\textit{while}(\text{b}) \text{ block}]\!]\sigma = \widetilde{\bigsqcup} \left\{ \Delta(\Gamma(\mathscr{S}[\![b]\!]_{false}(\sigma_i))) \;\middle|\; i \in \mathbb{N} \right\}$$

Since the abstract domain is finite (via the normalization form of Chapter 6) this set can easily be computed by iteratively processing the loop body until we reach a state that has already been seen (i.e. reach a fixpoint staring from bottom). At this point in time we know we have generated all the possible states that can appear in the loop body or at the exit of the loop.

9.2 Interprocedural Flow Algorithm

To efficiently analyze non-trivial programs we need to address several issues in the interprocedural analysis. Of particular concern are how recursive calls are handled and how the call graph is explored, how the analysis should deal with library calls (and other simple methods), and finally how to deal with the state size and exponential growth in the number of times each method is analyzed when performing a context-sensitive analysis.

9.2.1 Call Graph Exploration

The order in which methods are analyzed has a substantial impact on the efficiency of the program analysis [5, 44] and thus we use a hybrid approach to exploration that varies the order based on structural information about the call graph and the method bodies. In particular we distinguish between calls that are in a recursive cycle (a strongly connected component in the call graph) and calls that are non-recursive. In the non-recursive case we also want to account for the differences between builtin methods, small leaf (or *getter*/setter type methods) and more complex methods.

In the frontend analysis we identify small leaf methods and simple methods that either just access fields or call though to other methods. The frontend also computes a call graph using the declared type information which is used to drive the interprocedural analysis. As the efficiency of the interprocedural analysis is dependent on the precision of the call graph we are investigating alternative methods for the call graph computation. Once we have computed these properties of the program we use Algorithm 15 to analyze the program. In the descriptions that follow, we omit memoization support so that we can focus on the structure of the analysis algorithm. However, in practice, memoization of the analysis results needs to be performed to ensure acceptable performance.

Analyze Builtin. If the method that is being called is a builtin library method that has specialized semantics and these semantics apply (there are no *dependent* methods) then we can directly apply the specialized semantics (via *analyzeBuiltin*). The *dependent* method qualification allows specialized semantics for methods like `contains`, which depends on the definition of the `equals` method of the receiver argument. In many cases the `equals` method is the default reference equality definition from the `Object` class and we can precisely define the abstract semantics for the operation, but for other calls where the `equals` method is overridden we must analyze the implementation of the `contains` method body directly using the other analysis methods.

Algorithm 15: Analyze Interprocedural, (analyze m_{to} when called from m_{from} in abstract state σ)

input : program p, caller m_{from}, callee m_{to}, abstract state σ

if m_{to} *is a builtin method of p and no* dependent *methods* **then**
    p.analyzeBuiltin(m_{to}, σ);

else if m_{from}, m_{to} *in same recursive cycle of p* **then**
    $(\sigma_r, \sigma_u) \leftarrow$ project(σ);
    $\sigma_f \leftarrow p$.analyzeRecSame(m_{to}, stateMerge(σ_r));
    $\sigma \leftarrow$ extend(stateMerge(σ_f), σ_u);

else if m_{to} *in a recursive cycle of p* **then**
    $(\sigma_r, \sigma_u) \leftarrow$ project(σ);
    $\sigma_f \leftarrow p$.analyzeRecNew(m_{to}, stateMerge(σ_r));
    $\sigma \leftarrow$ extend(stateMerge(σ_f), σ_u);

else if m_{to} *is a small method* **then**
    p.analyzeDirect(m_{to}, σ);

else
    $(\sigma_r, \sigma_u) \leftarrow$ project(σ);
    $\sigma_f \leftarrow p$.analyzeDAGCall(m_{to}, stateMerge(σ_r));
    $\sigma \leftarrow$ extend(stateMerge(σ_f), σ_u);

Analyze Recursive. The *analyzeRecSame* method is used to analyze calls that are within the same recursive component of the call graph. For these calls we want the analysis to explore the call graph in a breadth first manner as opposed to the depth first exploration used for non-recursive calls. This avoids the problem of full path exploration in densely connected recursive call structures (a depth first exploration of all possible call paths in a strongly connected component of a call graph requires exponential time). Thus Algorithm 16 simply checks for the current approximate abstract semantics of the given abstract state (creating a new entry if needed) and returns this approximate result.

The *analyzeRecNew* algorithm is used for the first call into a recursive component of

Algorithm 16: analyzeRecSame

input : program p, callee m_{to}, abstract state σ

if m_{to} *has an approximation for the semantics of* σ **then**
    return *current approximate semantics for* σ *in* m_{to}

else
    Add m_{to} to worklist;

    $\sigma_{out} \leftarrow$ base case semantics for σ in m_{to};

    add (σ, σ_{out}) as approximate semantics pair to m_{to} ;

    return σ_{out}

the call graph. It sets up the worklists and then preforms the analysis of the methods and approximate abstract semantics until the fixed point is reached (thus safely approximating the result of the call). Algorithm 17 initializes a worklist with the called method and then proceeds to process the methods in this list in a breadth first fashion (methods are taken from the front of the worklist and added at the back in Algorithm 16 and in the conditional statement). While this simple worklist approach is reasonably effective it appears that the performance of the analysis can be improved significantly by altering the order in which the entries in the worklist are processed (although we have not been able to explore this topic in detail yet).

Analyze Direct. The analyze direct method allows the analysis to avoid the cost of project/extend (Section 9.3) and memoization for small calls where the cost of analyzing the method body is less than the cost of these operations. The simplest way to accomplish this is to inline these methods in the front end (which is what is currently done in the analysis). However this technique is limited by the ability to identify statically inlineable methods (i.e. they must have a single static target so dynamic methods cannot be handled) and extensive inlining can cause undesirable expansion of the code. To avoid this we propose using the front end to identify and uniquely rename the references in these methods so that the analysis can dynamically decide if the method body should be analyzed directly

Algorithm 17: analyzeRecNew

input : program p, callee m_{to}, abstract state σ

add $(\sigma, \bot)$ as approximate semantics pair to m_{to} ;

worklist $w \leftarrow [m_{to}]$;

while $w \neq \emptyset$ **do**
  $m \leftarrow w$.popFront();
  foreach $(\sigma_{\text{in}}, \sigma_{\text{out}})sp$ *in approximate semantics list of* m **do**
    $\sigma'_{out} \leftarrow m$.analyze($\sigma_{in}$);
    if $\sigma'_{\text{out}} \neq \sigma_{\text{out}}$ **then**
      w.addAll({methods $m' \mid m'$ calls m_{to}});
      sp.second $\leftarrow \sigma'_{out}$;
return *abstract semantics of* σ *in approximate semantics table of* m_{to}

as if it was inlined or to use the full set of calling operations as given in *analyzeDAGCall*.

Analyze DAG Call. The *analyzeDAGCall* method is used to analyze the larger methods that encapsulate significant functionality. The *analyzeDAGCall* simply passes the state into the local analysis method and returns the result. Analyzing of the callee before returning to the analysis of the caller results in the exploration of the call graph in a depth first order for non-recursive calls, which is an optimal order or these calls.

9.2.2 Call Entry/Exit Merge

The *stateMerge* operation allows the analysis to aggressively control the size of the states by computing a single model that is the summary of all the models in a state. The use of this operation as the regular join (or upper approximation) would result in unacceptable losses. However, we can take advantage of natural abstraction points to merge all of the elements in the projected set (after the project operation the models only refer to the fragment of the heap visible in the callee scope) into a single heap model. Intuitively methods

(particularly with non-trivial bodies or that are virtual) have strong abstractions for the portion of the heap that is passed in and properties that are important to the behavior of the method body either (a) hold for all possible program states at the call site, thus the property holds for the join or (b) the property only holds for some of the states but because of the weak assumptions made for the inputs to the method must explicitly test for this property before using it (e.g. the nullity test in a recursive list traversal). This assumption is confirmed by our experimental results which indicate that little or no information is lost by aggressively merging the input and return states from the analysis of method calls (also referred to as *multivariance* [44, 47]).

Algorithm 18 shows the pseudo-code for merging two abstract heap graphs. The full *stateMerge* algorithm can be constructed by pairwise applying this to the graph components and the scalar domain join to the scalar components of the models in the state. The graph merge algorithm is simply the union of the two graphs, assuming that two variable nodes with the same name are equal but all non-variable nodes are not, followed by the normalization of the resulting graph.

In practice it is possible to improve the precision of this operation by updating the node/edge merge operations to be more precise when joining nodes/edges that originate in different states. The two cases that we have found to be important are:

- When merging two nodes (edges) from two different models and both nodes (edges) have a *linearity* of 1 then the resulting node (edge) represents at most one object (reference) and should have a *linearity* of 1, not ω.

- When merging two edges, with the *non-interfering* property, from different models we know the pointers abstracted by the edges cannot be *connected* (since the two edges represent possibilities from two distinct models) and thus the resulting summary edge is also *non-interfering*.

Algorithm 18: stateMergeGraph

input : graph g, graph g',

output: graph g_m

$g_m \leftarrow g \cup g'$;

normalizeGraph(g_m);

return g_m

9.3 Project/Extend

A significant component of the analysis runtime is due to the need to perform a context-sensitive interprocedural analysis, where each procedure body may be analyzed multiple times (once for each different calling context).

The practice of using a memo-table to avoid recomputing analysis results and the use of a *project* operation to remove portions of the heap that cannot affect or be affected by the called procedure are standard techniques for minimizing the number of times each function needs to be analyzed during interprocedural dataflow [5, 6, 46, 47, 51]. The two major goals of the *project* operation are improving the effectiveness of memoizing analysis results by removing portions of the heap that could cause spurious inequalities between calling contexts and preventing the loss of precision that occurs when recursive procedures use a summary representation for multiple out-of-scope references (e.g. local reference variables with the same name but that exist in different call frames).

9.3.1 Abstract Call Stack

Our concrete model for the *call stack* is a function $S_m : (V \times \mathbb{N}) \mapsto O$, where V is the set of local variable names and $\mathbb{N}$ represents the depth in the call sequence (main is at depth 1) and O is the set of all live objects. Thus, the pair $(\mathsf{v}, 4)$ refers to the value of the variable v in the scope of the 4^{th} call frame.

To represent the concrete call stack we introduce *stack variables* which represent the values of local variables on the stack (for a variation on this approach see [57]). In our extension each *stack variable* summarizes all the possible targets (in a given graph) for a given variable name on the stack. Given a variable name `v` and a heap graph g we define a variable name `v'` for use in the abstract domain (we will select a better naming scheme in Section 9.4) where: `v'` is the abstraction of all the variables in the call stack, $\exists i \in \mathbb{N},$ node $n \in g,$ object o_n s.t. $o_n \in \gamma_f(n) \wedge S_m(\mathtt{v}, i) = o_n$.

By associating the set of stack locations that are abstracted with the set of targets in a given abstract heap graph, we can naturally partition the *stack variables* along with the heap graphs. Since each *stack variable* is associated with only the values on the stack that point into a region of the heap represented by the given heap graph, it is straightforward to partition and join them when partitioning the heap graphs, Algorithms 19 and 21.

Thus, during the local analysis the heap graph represents the portion of the program heap that is visible from the local variables and is augmented with some number of *stack variables* and *cutpoint variables* which relate variable values and the heap in the caller scope to the portions of the heap reachable from callee scope local variables.

For efficiency and in order to ensure analysis termination the naming scheme we choose will result in situations where multiple cutpoint (or stack) edges are given the same name. This may result in some amount of information loss (particularly with respect to reachability and aliasing). To minimize the loss that occurs we introduce an instrumentation domain for the stack/cutpoint variable edges, *nameColl* = $\{pdj, pua, pa\}$. Where *pdj* indicates a cutpoint/stack name representing (a single edge) or edges where the edges do not represent any pairwise *connected* references, *pua* indicates a name representing multiple edges where there are no pairwise *aliases*, while *pa* indicates a name representing edges that they may have pairwise *aliasing*. Thus, the cutpoint variable and stack edges are represented with records $\{$`id`, `linearity`, `interfere`, `connto`, `nameColl`$\}$.

9.4 Stack Variables, Cutpoint Labels

When performing the project operation in heaps with cutpoints we need to name the *stack variables* as well as the *cutpoint* edges. We use a simple technique for the stack variables: given a variable name `v` defined in the caller function `fcaller` we use the name `$fcaller*v` to represent this variable in the callee scope. This naming scheme can create false dependencies on the local scope names unless the variable information is anonymized during the comparisons of entries in the memo-table.

Naming edges that cross the cutpoints is more complex since we need to balance the accuracy of the analysis with the potential of introducing spurious differences resulting from isomorphic (or nearly so) cutpoint edges being given different names. For the renaming of the cutpoint edges we assume that special names for the arguments to the function have been introduced. The first pointer parameter is referred to by the special variable name `p1` and the i^{th} pointer argument is referred to by the variable `pi`.

For each cutpoint edge we generate a pair of names: one is used in the unreachable section of the heap graph and one in the reachable section, which allows an abstract heap model to represent both incoming and outgoing cutpoint edges that are identical and exist in the same abstract heap component without loss of precision.

If we are adding a cutpoint for the method call `fcaller` and the edge e, which is a cutpoint, starting at n and ending at n', and has edge label `fe`. We can find the shortest path (`f1` ... `fk`) from any of the `pi` variables to n' (using lexographic comparison on the path names to break ties). Using the `pi` argument variable and the path (`f1` ... `fk`) we derive the cutpoint `basename` = `fcaller*pi*f1*`...`*fk*fe` We compute a pair of static names (*unreachN*, *reachN*) where *unreachN* = `$basename-` and *reachN* = `$basename+`.

9.4.1 Project and Extend Algorithms

Project. We assume that before the *projectHeap* function is invoked all of the special argument variable names have been added to the heap model. This allows *projectHeap* (Algorithm 19 below) to easily compute the section of the heap model that is reachable in the callee procedure and then compute the set of nodes that comprise the unreachable portion of the heap model.

Algorithm 19: projectHeap

input : g: the heap model to be partitioned

output: g_r, g_u: the reachable and unreachable partitions, *snu*, *ncs*: the static names used and newly created

reachNodes ← set of nodes reachable from args;

unreachNodes ← set of nodes unreachable from args;

crossEdges ← set of edges that start in *unreachNodes* and end in *reachNodes*;

snu ← ∅;

ncs ← ∅;

foreach edge *e* in *crossEdges* **do**
    (sn, isnew) ← procCrossEdge(*g*, *e*, *reachNodes*);
    snu.add(*sn*);
    if *isnew* **then** *ncs*.add(*sn*);

h_u ← subgraph of g on the nodes *unreachNodes* ∪ {dummy nodes from procCrossEdge};

h_r ← subgraph of g on the nodes *reachNodes*;

return *(*g_r, g_u, snu, ncs*)*;

For each edge that crosses from the unreachable section into the reachable section we add a pair of static names to represent the edge (Algorithm 20). Since the heap model stores a number of domain properties in each edge, we create a dummy node and remap the edge to end at this node. Then, the *unreachN* static name is set to refer to this dummy

node. In the reachable portion of the heap graph we simply set the *reachN* static name to refer to the target of the cross edge.

When adding the *reachN* static name to the reachable section of the heap graph the name may or may not already be present in the heap graph. If the name is not present then we add it to the static name map and for later use we note that this is the call where the name is introduced. Otherwise a name collision has occurred and we must mark the edges representing the possible cutpoints appropriately (for simplicity we mark all the edges). If there may be aliasing we note that the cutpoints from different frames may have aliasing targets (*pa*) and similarly if the new cutpoint edge may be connected with an existing cutpoint edge we mark them as being pairwise connected (*pua*). The functions *makeEdgeForUnreachCutpoint* and *makeEdgeForReachCutpoint* are used to produce edges to represent the cutpoint (based on the static name and the cutpoint edge properties) in the unreachable and reachable portions of the heap.

Once all of the cutpoint edges have been replaced by the required static names, the heap can be transformed into the unreachable version (where all the nodes in the reachable section and all the variables/static names that only refer to reachable nodes have been removed) and the reachable version (where the nodes in the unreachable section and the associated names have been removed).

Extend. After the call return we need to rejoin the unreachable portion of the heap that we extracted before the procedure call entry with the result we obtained from analyzing the callee procedure. This is done by looking at each of the static names that was used to represent a cutpoint edge and reconnecting as required. Then, each of the newly introduced cutpoint names can be removed from the heap model. The pseudo-code to do this is shown in Algorithm 21.

This algorithm merges all edges with the same reachable cutpoint name so that there is at most one target edge for a given cutpoint name in the reachable heap graph g_r (this

Algorithm 20: procCrossEdge

input : g: the heap, e: the cross edge, *reachNodes*: set of reachable nodes

output: *rsn*: the name used, *isnew*: true if *rsn* a new name

$n_e \leftarrow$ the node e ends at;

$n_i \leftarrow$ new dummy node;

(ursn, rsn) $\leftarrow$ genStaticNamePairForEdge(h, e);

$e_u \leftarrow$ makeEdgeForUnreachCutpoint(e, *ursn*);

set endpoint of e_u to n_i;

add e_u as an edge for *ursn*;

$e_r \leftarrow$ makeEdgeForReachCutpoint(e, *rsn*);

set endpoint of e_r to n_e;

remap the endpoint of e to n_i;

if the name *rsn* exists and has edges pointing to a node in *reachNodes* **then**

  rsnes $\leftarrow \{e' | e'$ is an edge for the cutpoint var *rsn*$\}$;

  add e_r as an edge for *rsn*;

  if e_r *is connected with an edge in* rsnes **then** set edges in *rsnes* and e_r to *pua*;

  if e_r *may alias with an edge in* rsnes **then** set edges in *rsnes* and e_r to *pa*;

  return *(*rsn, false*)*;

else

  add the name *rsn* to h;

  add e_r as an edge for *rsn*;

  return *(*rsn, true*)*;

simplifies the algorithm and is in our experience is quite accurate). The algorithm then pairs up the two cutpoint names and remaps the edge we saved in the unreachable section to the target node in the reachable section subject to a number of tests to propagate sharing information (the nullity information is propagated due to the fact that the dummy node and all incoming edges are always removed but the foreach loop on the targets of *ursn* does not execute since the target set is empty). The e_r*.nameColl = pua* test is true if this edge

represents sets of pointers that do not have pairwise aliases. Thus, we mark the newly remapped edge and e_r as pairwise unaliased. Similarly, the e_r.*nameColl* = *pdj* test is true if this edge represents cutpoint/stack edges that are pairwise disjoint. Thus, we mark the newly remapped edge and e_r as pairwise disjoint.

Algorithm 21: extendHeap

input : g_r, g_u: the reachable and unreachable partitions, *snu*, *ncs*: the static names used and newly created

output: g: the joined heap model

$g \leftarrow$ new *heap*();

g.heapGraph $\leftarrow$ mergeGraphs(g_r.heapGraph, g_u.heapGraph);

foreach static name *sn* in *snu* **do**
    ursn $\leftarrow$ reachNameToUnreachName(*sn*);
    $n_r \leftarrow$ the target of *sn* in g_r.nameMap;
    foreach node n_u that is a target of *ursn in* g_u.*nameMap* **do**
        $e_r \leftarrow$ the single incoming edge to n_u;
        remap e_r to end at the target of n_r;
        e_r.interfere = e_r.interfere $\sqcup$ n_r.interfere;
        if e_r.nameColl = *pua* **then** set e_r and n_r as unaliased;
        if e_r.nameColl = *pdj* **then** set e_r and n_r as disjoint;
    h_u.removeNodeAllEdges(target of *ursn*);
    h_u.unmapStaticName(*ursn*);
    if sn *in* ncs **then** h_r.unmapStaticName(*sn*);
g.nameMap $\leftarrow$ mergeNameMaps(g_r.nameMap, g_u.nameMap);

return g

The major components of this algorithm are the separation of the *mergeGraphs* action from the *mergeNameMaps* action and the elimination of the static cutpoint edge names that were introduced for this call.

The *mergeGraphs* function computes the union of the graph structures that represent

the abstract heap objects, while the *mergeNameMaps* function computes the union of the name maps (which are maps from the stack/variable/cutpoint names to the nodes in the graph structure that represent them). This separation allows the algorithm to nullify the names created for this call which prevents the propagation of unneeded cutpoint edge targets to the caller scope. The function *unmapStaticName* is used to eliminate a given static name from the abstract heap model name map.

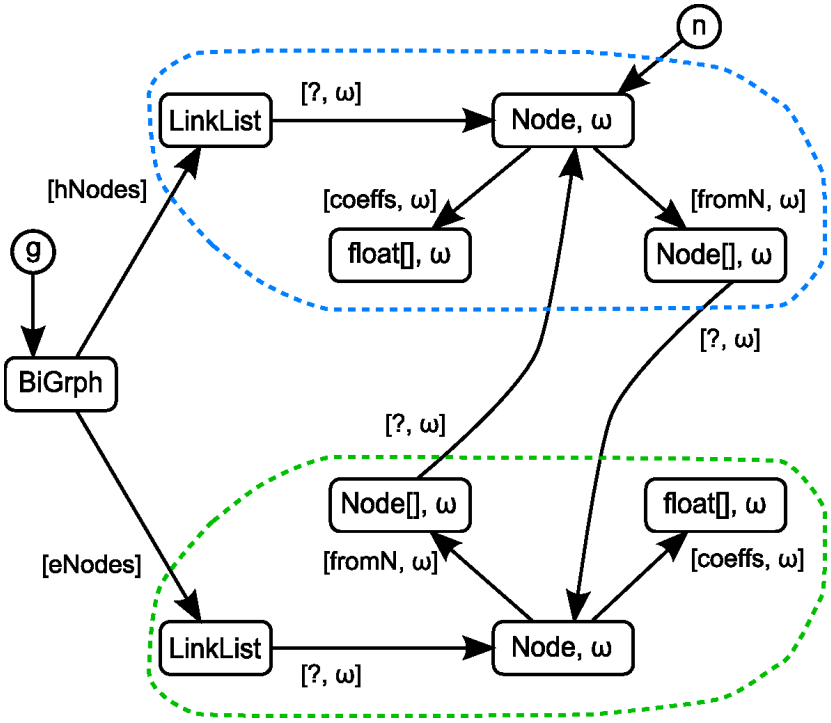
9.5 Example Project/Extend.

In Figure 9.1 we show an example of how the project operation functions, how it reduces the size of the abstract models that need to be processed and reduces the number of times each method needs to be analyzed. In Figure 9.1(a) we show a simplified version of the state of the program heap at the call to the `computeNewValue` method in the `em3d` benchmark. The call takes a `Node` object referred to by the variable `n` and computes a new electric/magnetic field value for it.

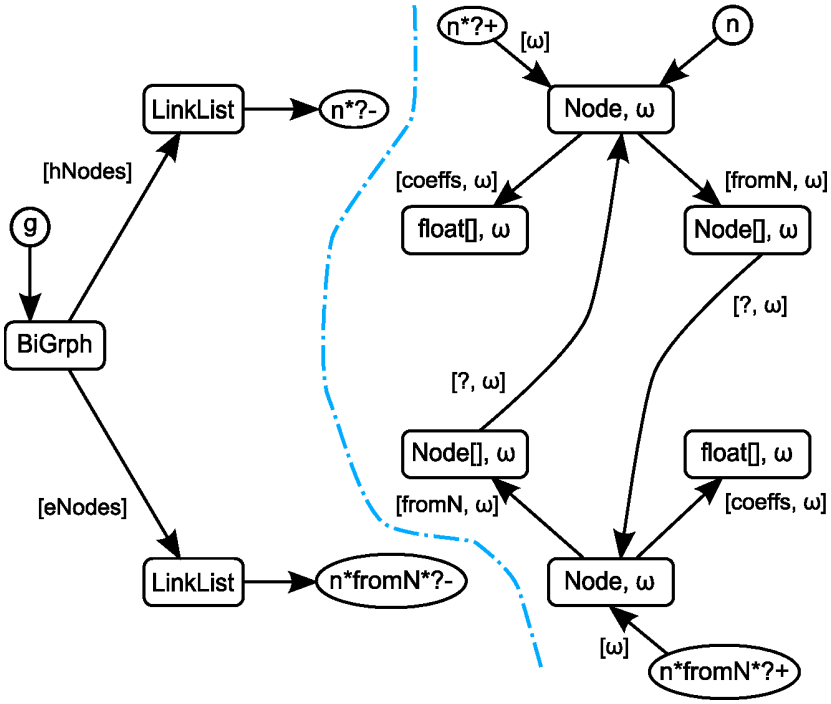
By using the project operation we can partition the heap into the section of the heap that is used by the `computeNewValue` method from the section that cannot affect (or be affected) by the method. This partitioning is shown in Figure 9.1(b) (we use the dashed, blue if color is available, line to show the partition). We have added the special cutpoint variables `n*?-`, `n*?+` and `n*fromN*?-`, `n*fromN*?+` to the unreachable and reachable sections of the heap to represent the edges that are cut during this partition. These names will allow the analysis to correctly reconnect these edges on return from the method.

The result of this transformation is that we have reduced the number of non-variable nodes by 3 (a 30% reduction in the size of the heap model). We have also eliminated spurious calling context information which will allow the analysis to avoid reanalyzing the procedure later. In this example the `computeNewValue` method is also called

on `Node` objects from the electric field. Without the project operation the differences that arise from the field names `hNodes` vs `eNodes` will force the re-analysis of the `computeNewValue` method. However, the project operation results in identical models (the heap graph on the right hand side of Figure 9.1(b)) being passed to the local analysis procedure for both call sites. Since the program states are equivalent at both call sites the analysis can use the memoized results produced from analyzing previously.



(a) Call to `computeNewValue`



(b) After Project

Figure 9.1: Project/Extend for `computeNewValue` in em3d

Chapter 10

Semantics of Collections and Libraries

Library-based collections are a fundamental component of modern programming languages and are used extensively in almost any non-trivial program. Substantial work has gone into developing heap analysis tools that can accurately and efficiently analyze simple data structures, mainly lists, trees, and simple cyclic structures. Unfortunately, most existing heap analysis techniques have aspects that make their use in analyzing large programs that use standard libraries impractical. This is either due to the inability to model the complex data structures (red-black trees, doubly-linked lists with tail pointers, etc.) used in the library code or due to the computational complexity of performing the analysis of these complex library data structures. An alternative to directly analyzing the code that implements the collection objects is to use the semantics of the collection objects to simulate the effect of each collection operation as an atomic program operation.

In addition to the performance issues that arise when directly analyzing the collection library implementations, the semantics based approach allows the modeling of properties specific to each collection type (e.g. sets never contain duplicate elements).

This chapter presents a method for representing the semantics of collection libraries and iterators over the collections in our shape analysis framework. The representation that

we present for the collection semantics enables the shape analysis to identify individual elements in the collection, allowing them to be strongly updated. The iterator semantics provide a representation for the notion of progress in the processing of the elements in the collections, which allows the shape analysis to accurately model the processing of the collections.

For this chapter we are going to use a simplified set of collections and iterator operations to simplify the discussion and to separate out the semantics of the *advance* and *get* operations (as they are combined in the semantics of the `next` operation in the Java standard library which advances the iterator and fetches the argument as an atomic operation). This simplification allows us to focus on the semantics of the individual components and can be easily adapted to the semantics of the collections in the standard Java libraries.

10.1 Example Programs

To gain some insight into how our extensions work and interact with the base heap analysis we use the examples in Figure 10.1. The examples use objects of types, `t1` and `t2`. The `t1` type has a single field `val` that points to objects of type `t2`. The `t2` type is a simple object with no pointer fields. The first code segment is a loop that fills a `set` with objects of type `t1` (all of which have a pointer to the same object in the `val` field). The second example takes the resulting `set` and updates each element to point to the `t2` object that the variable `r` points to.

We are using the `t1` and `t2` types to keep the examples simple. However, the methods presented here can handle similar programs, with the same level of accuracy, where `t1` and/or `t2` are replaced by simple finite structures, lists, trees, or other library collections. The analysis algorithm is also able to analyze our examples when `t1` and/or `t2` are replaced with DAG shaped or cyclic structures, although potentially with reduced accuracy.

Initialize a Set

```
set p = new set()
t1 q
t2 s = new t2()
for(int i = 0; i < M; ++i)
    q = new t1()
    q.val = s
    p.insert(q)
```

Update all the elements in the set

```
t2 r = new t2()
iterator i = p.begin()
while(i.isValid())
    (i.get()).val = r
    i.advance()
```

Figure 10.1: Example Code

In both examples the analysis should determine that every element in the `set` is unique (although the elements may reference the same object in the `val` field). In the second example the analysis should capture the fact that on each iteration of the loop the element that the iterator refers to has its `val` offset updated and after the loop all the elements in the `set` have been updated. Thus, there are no longer any objects in the set with pointers in the `val` field that refer to the same object as the variable `s`.

10.2 Domain Extensions For Collections

The fundamental idea for modeling the collections and iterators is to classify the pointers that are stored in a collection into four categories based on their relation to any iterators that are acting on the collection. Based on this classification we create a special *offset* for each category, just as was done for arrays.

- Pointers that have an unknown relation to the active iterator or when there is no active iterator for this collection. Edges representing pointers in this category are given the label *?*.
- The single pointer that the iterator is currently at in the collection. The edge repre-

senting this pointer is given the label *at*.

- Pointers that come before (in whatever iterator order is specified by the collection) the location that the iterator is at. Edges representing pointers from this class are given the label *bi*.

- Pointers that come after (in whatever iterator order is specified by the collection) the location that the iterator is at. Edges representing pointers from this class are given the label *ai*.

This scheme for classifying the pointers in a collection is a specific case of the *partitioning functions* that are used in [22] to partition arrays of scalars. The definition we use is only precise when there is a single iterator that is active in a collection. In the case of multiple iterators simultaneously indexing through a collection our partition must conservatively assume that any relation could hold between the positions of the iterators. The use of more flexible *partitioning functions* would allow our analysis to partition the pointers in a collection even when multiple iterators are being used to index through the collection. However, the use of more general *partition functions* substantially complicates the analysis and we expect that most of the time only a single iterator will be active in a collection. Based on this assumption we opted for the fixed partition.

Modifications to the Dataflow Operators. Our modifications have only a minimal impact on the algorithms for the abstract semantics and we only need to modify the node merge algorithm. First, we define a simple function that takes a node and if it is currently partitioned on an iterator forgets all the partition and iterator information. The procedure to *forget* this information is shown in Alg. 22.

Algorithm 22: clearIndexer

input : g a heap graph, n a node

if *n has an active iterator* **then**
    n.activeIter $\leftarrow \perp$;
    n.empty $\leftarrow$ *unknown*;
    n.atFirst $\leftarrow$ *false*;
    foreach *out edge e* **do**
        if $e.offset \in \{\text{bi}, \text{at}, \text{ai}\}$ **then** $e.offset \leftarrow ?$;
g.removeEquivalentEdges(n);

10.3 Modeling Iterator and Collection Operations

In this section we look at how the various collection methods are implemented. Even our simplified collection library has a non-trivial number of methods to manipulate the various collection objects and the associated iterators. Thus, we focus on describing the most interesting methods. For simplicity we assume that all of the collection nodes have a *linearity* of the 1. If the linearity is ω the default action is usually simply set the *empty* property to *unknown* and no-op, in the cases where the action is more interesting we will explicitly mention it.

Insertion and Deletion. For the insert operation we first call the *clearIterator* method. Next we add an edge from the collection to the object that we want to add to the collection and we set the label of this edge as *?* and if the *linearity* property is 1 then we also know that the collection has at least one element and can set the *empty* property to *false*. If the node has *linearity* of ω we do not know which of the many collection objects may have been added to so we conservatively assume that any of them may have been inserted into and set the *empty* property to *unknown*.

The *delete* operation for our collection library takes an iterator and removes the element referred to by that iterator from the collection. To model this we remove the edge

with *at* label (which strongly deletes the iterator target from the collection). This operation may or may not make the collection empty so we must conservatively set the *empty* property to *unknown* and the *atFirst* property to *false*.

Clear, Empty and Size. The abstract semantics for the `clear`, `empty` and `size` operations are all relatively straight forward. For the `clear` operation we simply remove all outgoing edges (thus simulating the removal of all the references in the container) and set the *empty* property to *true*. For the `size` operation on a given collection (suppose `x = y.size()`) we add the equality relation $x = card(y)$ to the domain of uninterpreted function symbols and if we know that the collection `y` refers to has the *empty* property is *true* then we have the stronger condition $x = card(y) = 0$.

The semantics for the `empty` method are slightly more complex as we want to generate one model for then the method returns *true* and one model for when the method returns *false*. If we can determine that the collection *must* be empty/non-empty then we can just return (if the *empty* property is either *true* or *false*). In the case where the collection *may* or *may not* be empty we create a model for each case. The code to do this is shown in Algorithm 23, we assume that the method *copy* creates a copy of the given program model, while *getTarget* returns the node that a variable refers to in the model and *addEquality* updates the abstract domain of equalities with the new equality information.

Iterator Initialization and Get. The most common way to initialize an iterator is to get an iterator to the first element (with respect to the collection's iteration order) of a collection. The `begin` method in our collection library is used to do this. To simulate the effect of this operation in the heap graph (Algorithm 24) we use the *clearIterator* method to forget the partitioning of any other iterators on the collection. Then we split each possible summary edge ($?$) into two new edges (accomplished by the *splitIthSummary* method from Chapter 8): one is used to represent the element in the collection that the iterator refers

Algorithm 23: Empty/Non-empty Materialization

input : m model, c variable referring to (collection) node of linearity 1

output: m_e empty model, m_n non-empty model

$m_e \leftarrow m$.copy();

m_e.getTarget(c).clearOutEdges();

m_e.getTarget(c).empty $\leftarrow$ true;

m_e.addEquality(*card*(c) = 0);

foreach *variable v, refers to n* **do**
    addEquality($card(v) = 0$);

$m_n \leftarrow m$.copy();

m_n.getTarget(c).empty $\leftarrow$ false;

return (m_e, m_n)

to (*at*), the other edge is used to represent all the elements that come after the element referred to by the iterator (*ai*). Since the iterator must refer to the first element in the collection (with respect to iteration order) we do not need an edge to represent elements that come before the iterator. Depending on the *interfere* property of the *?* edge we set the split edges as being *disjoint* (if the edges was *np*), *connected* (if the edges was *ip*) or *aliasing* (if the edges was *ap*). We also know that the iterator is at the first element in the collection so we set the *atFirst* value to *true*.

The `get` operator can be treated as a simple field load off the special field *at*. Using this approach passes all the work onto the existing abstract heap graph load framework which performs the appropriate operation. After a call to the get operation we can also infer information about the emptiness of the collection object. The get operation only succeeds if the collection is non-empty and thus we can update the *empty* property to the *false* value.

Algorithm 24: iteratorBegin

input : model m, var c (refers to collection of linearity 1), iterator var i

output: model[] r

$n \leftarrow m$.target(c);

n.clearIndexer();

$r \leftarrow$ new model[n.summaryCount()];

for $k \in 0 \ldots r$.Length $- 1$ **do**

$m' \leftarrow m$.copy();

$n' \leftarrow m'$.target(c);

r[k] $\leftarrow m'$.splitIthSummary(k, c, `yes`);

n'.atFirst $\leftarrow$ *true*;

Iterator Advance. After initializing an iterator we often want to advance it through the collection (the `advance` method) and use the `isValid` test to check if the iterator still refers to a valid point in the collection.

The advance method needs to re-label the existing edge with the *at* label to have the *bi* label and create a new edge with the *at* label that is parallel to the edge with the *ai* label if such an edge exists (accomplished by the *splitIthAfterIndex* method from Chapter 8). This is shown in Algorithm 25, which assumes that the given iterator is valid and is the current active iterator for the collection. We also update the *atFirst* value to *false*. Further, we can infer that since the `advance` method succeeded that they collection has at least 1 element in it and is thus non-empty (which allows the analysis to set the *empty* property to *false*).

IsValid. In the `isValid` method we want to (when possible) propagate the knowledge that on a given path `isValid` returned *true* or *false* and update the model to represent this information. If we take a branch that can only be executed when a given iterator is invalid then we want to update our model to reflect this information (Algorithm 26). To do this we have two cases. If the given iterator is not the active iterator we do nothing. If the

Algorithm 25: iteratorAdvance

input : model *m*, var *c* (refers to collection of linearity 1), iterator var *j*

output: model[] r

$r \leftarrow$ new model[*n*.afterCount()];

for $k \in 0 \ldots r$.Length $- 1$ **do**

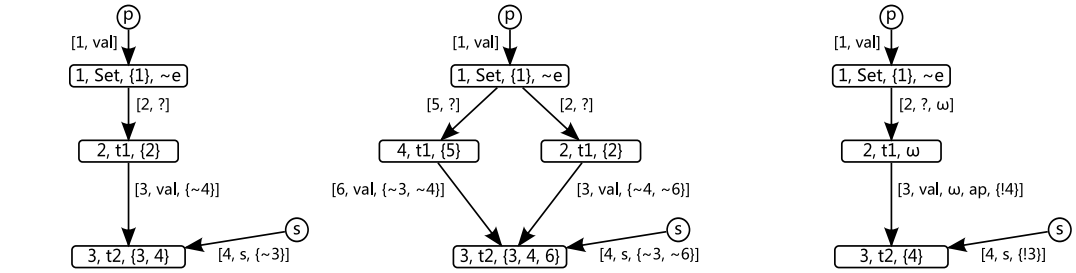
$m' \leftarrow m$.copy();

$n' \leftarrow m'$.target(*c*);

r[k] $\leftarrow m'$.splitIthAfterIndex(*k*, *c*, *j*);

n'.atFirst $\leftarrow$ *false*;

n'.empty $\leftarrow$ *false*;



(a) Added First Entry (b) Added Second Entry (c) Normalize

Figure 10.2: Add Elements to a Set Container

given iterator is the active iterator we delete the edges with the *at* label and the edges with the *ai* label. Further, if *atFirst* if *true* and `isValid` returns false then we know that the collection must be *empty* and can update the *empty* property to *true* (and the cardinality properties with an equivalence to 0). The *eraseEdgeWithOffset* removes the edge with a given offset from the abstract heap graph and any nodes the edge *dominates* along with any other edges that are *dominance equal* (just like in the pointer comparison operations). Our current abstraction has no way to represent that an iterator must be valid so in the case that `isValid` only updates the *empty* property to *false* (since if the iterator *isValid* there must be at least 1 entry in the collection) and returns *true*.

Algorithm 26: isValid$_{false}$

input : i an iterator

$n \leftarrow$ the target of i;

if *i is not the active iterator for n* **then** return;

n.eraseEdgeWithOffset(*at*);

n.eraseEdgeWithOffset(*ai*);

if *n.atFirst* **then**
    n.empty $\leftarrow$ true;
    foreach *variable v, refers to n* **do**
        addEquality($card(v) = 0$);

10.4 Examples

Initialize the Set Figure 10.2(a) shows the abstract domain at the end of the first loop iteration. The variable `p` points to the `set` object and the variable `s` points to the object of type `t2` that all the elements in the set will reference. The first of the `t1` objects has been allocated and has had the `val` field set. Since we just allocated the object that the node represents we know it has a *linearity* of 1 and a *(S)ingleton* layout.

We also created an edge from the `set` object to the `t1` object. Since this edge was just created (by a store to an unknown location in the collection) it must represent a single pointer stored in the collection (*linearity* $= 1$, *interfere* $=$ *np* and *offset* $=$?). Finally, since we know that the set contains at least one element we have marked it as being *non-empty* ($\sim$e).

Figure 10.2(b) shows the state of the heap model at the end of the second iteration. Another element has been allocated and inserted into the set. The `val` offset of this object has been set to refer to the same node that `s` points to. Since there are now two incoming edges that may be connected and from the assignment statements we know they are both equal to `s` all three edges are pairwise *domEQ* and all *dominate* the node.

Since the abstract heap in Figure 10.2(b) is not in normal form (the `set` node has *equivalent* edges) it needs to be normalized (Chapter 6). This results in the abstract heap in Figure 10.2(c).

The two nodes with type `t1` have been combined into a new summary node. The edges with the labels *?* have been joined and are represented by an edge labeled [?, *ω*, *np*] since the edge represents more than one pointer and the pointers cannot interfere. Finally, the edges with the labels `val` have been joined and are represented by an edge that is labeled [`val`, *ω*, *ap*] since the edge represents more than one pointer and the pointers may *alias* (the edges that were joined had the *share* relation). Running through the loop again produces the same result, thus we have covered all possible iterations of the loop and are done.

Update the Set The second example from Figure 10.1 traverses all elements in the `set` (from the first example) and updates the `val` field of each object to refer to the same object as `r`. Figure 10.3(a) shows the state of the abstract heap after allocating a second object of type `t2` and initializing the iterator. We have set the iterator `i` to point to the `set` object, created a new edge to represent the single entry the iterator refers to (the edge with label *at*) and a new edge to represent the entries that come later in the iteration order (the edge with the label *ai*). Since we just initialized the iterator we know it refers to the first element in the collection and thus mark it as being at the beginning `-B` in the node label. When initializing the iterator the unknown edge *?* is *np* which means that the newly created edges (*at* and *ai*) can not be *connected*. Thus, the refinement method can split the node that represents the `t1` objects into two nodes (one representing the heap reachable from the *at* edge and one representing the heap reachable from the *ai* edge). Additionally, the *at* edge has *maxCut* of size 1 and points to a *(S)ingleton* node, thus the refinement algorithm can safely assume that the target has *linearity* 1 as well.

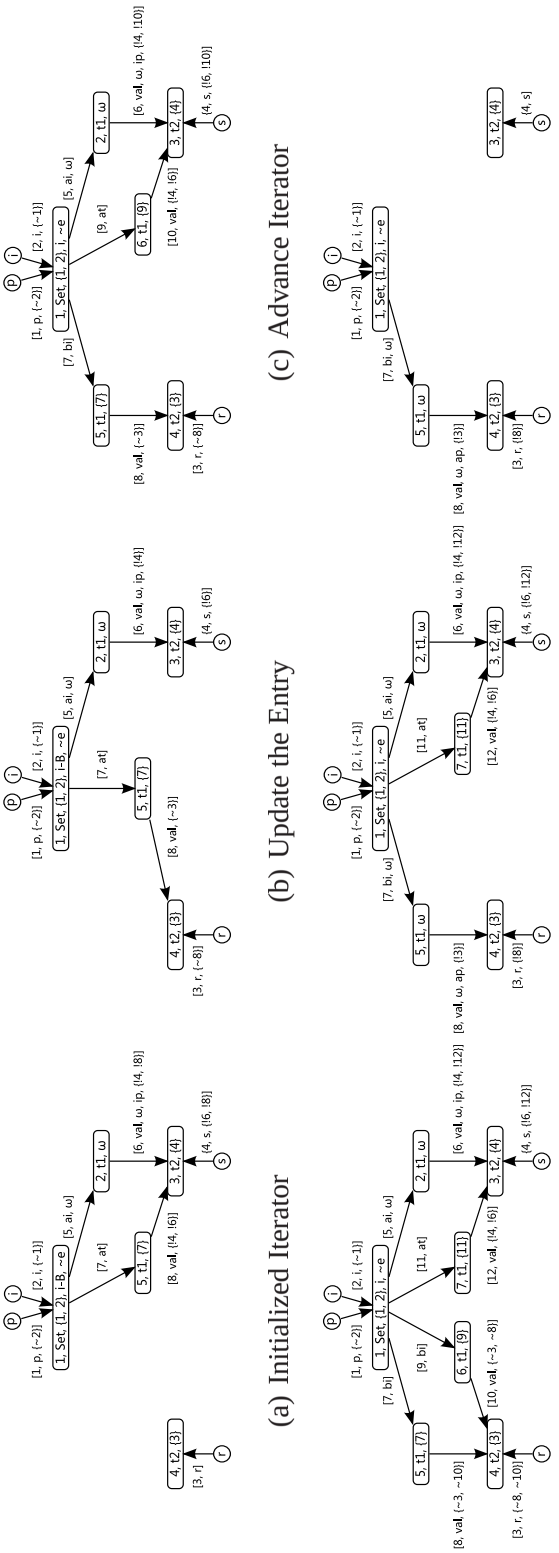
This allows the node to be strongly updated when the assignment is done. The result is

shown in Figure 10.3(b). When the iterator is advanced we set the current *at* edge to have the label *bi* and split a new out edge from the current *ai* edge. The result of this is shown in Figure 10.3(c), which is the state of the abstract heap at the end of the first abstract loop iteration.

The state of the heap model at the end of the second iteration is shown in Figure 10.3(d). The assignment was able to strongly update the target of the `val` field of the object referred to by the iterator. The iterator advance has indexed the current iterator position, splitting out a new *at* edge and resulting in two edges with the label *bi*. Thus, we need to combine their targets into a summary node and join the edges. This results in the abstract heap shown in Figure 10.3(e).

In Figure 10.3(e) we have some unknown number of pointers before the current iterator which all point to unique objects of type `t1` (the edge is *np*) and each of these objects has a reference stored in their `val` field, which (may) point to the same object as the variable `r`. Then we have the single element currently referred to by the iterator and some number of pointers that come after the iterator, which refer to the objects that have not been updated. The state shown in Figure 10.3(e) is also the repeated state of the abstract loop execution so we are done processing the loop body.

If we apply the exit test condition, `isValid`, which erases the edges with labels *at* and *ai*, to the state shown in Figure 10.3(e), we get the result shown in Figure 10.3(f). Note that there are no longer any references from the objects in the `set` to the region of the heap pointed to by `s`: each element in the set was strongly updated and by modeling the progress of the iterator we determined that the contents of the collection have been strongly updated.



(a) Initialized Iterator (b) Update the Entry (c) Advance Iterator

(d) End of Second Iteration (e) Normalize and Fix Point (f) Interpret Exit Test

Figure 10.3: Update Data in the Set

10.5 Extensible Modeling of Library Code

In order to efficiently and precisely analyze the rich and extensive standard libraries that appear in Java and other modern programming languages we want to build a high level facility for describing the semantics of a given builtin module that enables us to easily add new modules and minimizes the number of semantic operations we need to hard code into the analysis. Thus we extended the Java language with a set of special builtin annotations to support specifying which methods belong to a builtin class and how each of these methods should be processed.

The annotations appear after the standard Java method signature definition. We first have a specifier that allows us to determine if we want to provide a direct implementation of the method in Java that the analysis will process directly or if we want to specify a particular hard coded semantic operation (*semop*) in the analysis that should be used to determine the effect of the method on the abstract state. This allows us to support precise and efficient operations for commonly used and semantically nuanced operations without requiring us to implement the semantics of every library operation directly in the analysis (which is a time consuming and error prone task).

Many of the collection libraries that we are interested in modeling implicitly make use to other fundamental operations (in particular `equal`, `compareTo`, `hash`) that for most invocations use the default values from their definitions in the `Object` class but sometimes use overridden definitions of these operations. Thus, we need to ensure that this is handled by our builtin semantic definitions. One option is to support calls from the builtin semantics back out to the analysis semantics (similar to programming languages allowing calls both from interpreted code to compiled code and compiled to interpreted). This however can be difficult to implement and has many opportunities for introducing errors into the analysis. Instead we add another specifier (*overrideop*) field that allows us to use a builtin semantic operation when the default values of a given operation are used

(i.e. when the receiver argument has the default semantics for the `equal` operation we use the builtin semantics for the `contains` method) and then to directly analyze a simple Java implementation of the method if the default semantics of the operation are overridden (i.e. if the object overrides the `equal` operation to analyze a simple loop that implements the semantics of the `contains` method).

Our extended Java grammar then supports the additional annotations as follows on each method definition (where *semop, overrideop* are optional and if a *semop* is given then *body* is optional as well:
javadecl @ *semop* & $overrideop_1 \ldots overrideop_k$ *body*

Chapter 11

Read/Write Dependencies

The concept of data dependence between program statements is a fundamental tool for the reordering of program statements and the determination of invariant values in basic blocks, loops, or methods. Knowledge of data dependence allows the introduction of instruction-level parallelism and thread-level parallelism (both in loops and method invocations). In past work effective techniques for computing data dependence between scalar variables have been developed. However, the extension of this work to tracking memory-carried data dependence has been much less successful, in large part due to the lack of suitable heap analysis techniques to support them.

Previous work focused broadly on two approaches for identifying heap-carried data dependence, shape analysis as a proxy for data dependence [21, 29, 58] wherein the identification of various acyclic structures is used to infer which expressions cannot access the same portion of the heap, and the explicit tracking of modified locations [14, 34, 35] which model the set of locations that may be read/written at each program point. This early work introduced several fundamental concepts involved in explicitly modeling heap-carried data dependence. However experimental work with these approaches was limited to small numbers of micro-benchmarks. Thus, they do not address several important tech-

nical problems that arise when attempting to analyze the heap-carried data dependence problem for non-trivial programs.

Our analysis technique uses an explicit store model for the heap objects which allows us to easily track the identity of objects between program statements. This differs from some recent work on shape analysis, which uses logical models with implicit store representations [25, 59] that cannot be efficiently extended to track the properties of arbitrary heap locations. It also differs from approaches based on separation logic which restrict the program to regular recursive structures and limited sharing of objects on the heap in order to ensure termination [3, 4, 26]. These features preclude the use of these approaches on many realistic application programs including the `em3d` and `bh` benchmarks, which we analyze as detailed case studies here.

11.1 Running Examples

The running example creates 2 `Data` objects, each of which has a single integer field `val`, and puts them in a `Pair` object. If the conditional holds the `first` element of the pair is modified and then the swap method is called to interchange the `first` and `second` elements of the pair. This example is simple but relevant since in order to determine that the asserted property always holds the analysis needs to be able to track how pointer stores affect reachability relations in the heap, to identify where each heap location may be written, and do so across method invocations.

11.2 Data Dependence Extensions

To track the read/write histories of objects on the heap we extend the model presented in Chapter 3 with information to track the identity of the objects represented by a given node,

```
m1  void main() {
m2     Pair p = new Pair(new Data(5), new Data(10));
m3     if(*)
m4       p.first.val = 0;
m5     swap(p);
m6     assert(p.first.val != 0);
m7  }

s1  void swap(Pair p) {
s2     Data temp = p.first;
s3     p.first = p.second;
s4     p.second = temp;
s5  }
```

Figure 11.1: Conditional Modify and Swap

and for each field in the object we track the *most recent* program location (statement or control flow structure) where a read/write of that field *may* have occurred.

In order to ensure that the initial shape analysis when augmented with the read/write domain remains efficient it is critical to minimize the amount of additional information that is added to the heap model. The key observation is that for most optimization applications the shape analysis only needs to provide precise information about the *most recent* program location at which each field *may* have been read or written. Thus, the analysis does not need to track every possible program location where a field may have been read/written, and this significantly reduces the computational requirements. For the remainder of this chapter we assume that each statement and each control flow structure in the MIL program (Chapter 2) we are analyzing has a program location ℓ associated with it.

11.2.1 Extended Domain

Read-Write Locations. Each node may represent a number of objects of different types ($\tau_1 \ldots \tau_m$) and each type may have many fields ($f^1_{\tau_i} \ldots f^n_{\tau_i}$). For each of these fields we keep two program locations (ℓ), the last time the field *may* have been read (ℓ_r) and the last time the field *may* have been written (ℓ_w).

Node Identity. In order to efficiently analyze method invocations we memoize the input and return abstract states and reuse them as possible. In order to prevent spurious inequalities between the read/write program locations (that refer to the locations in the caller scope) in the memoized models we replace them with a generic *modified outside* value. To allow us to match the identities of the objects in the input state with their position in the output state we add a unique identity tag (a value in $\mathbb{N}$) to each node that is passed into a method call.

In our extended domain each node in the heap is now represented as a tuple `[type, linearity, layout, scalar-fields, identity]`. The entries `type`, `layout` and `count` are as described in Section 3. The `scalar-fields` entry is a list of `field-readloc-writeloc` entries, one for each scalar field, where `readloc` and `writeloc` are either a program location ℓ or the special entry 0 (*modified outside*). The `identity` entry is a *set* of identity tags or is omitted entirely if the node does not have an identity tag associated with it (or for clarity if it is not relevant to the example).

To track the read/write information for the pointer fields we extend each edge label to `[offset, linearity, interfere, readloc-writeloc]` where `readloc` and `writeloc` are defined the same as for the scalar fields in the nodes. Again, for clarity, we omit `readloc-writeloc` information if it is irrelevant to the example.

Figure 11.2 shows the (simplified) model that is computed as the result of executing the pair constructor in the first example program. The pair is marked as having read and written the two pointer fields at initialization (the `m2-m2` entries on the `first` and `second`

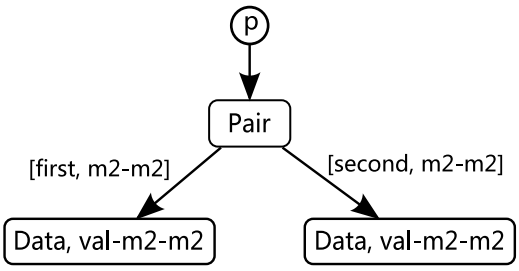


Figure 11.2: Simplified Model of `Pair` with *use-mod*

edges) and the identity tag is omitted (since this object was allocated in the current scope). The two `Data` objects which had their *val* fields initialized at program location *m2* have the entry `m2-m2` in their *scalar-fields* read/write entry.

11.2.2 Local Data Dependence

Now that we have extended the model with the required instrumentation properties we can define a set of dataflow operations to model the effects of program operations on the read/write information. The changes for load and store operations are simple, only requiring an update of the last read/write value for the target object to the current program location. Thus we omit a detailed description of these operations.

Abstract Conditional Semantics. To avoid tracking multiple models that differ only from minor differences in the *readloc-writeloc* locations we replace all the *readloc-writeloc* entries that refer to program locations in the *true* or *false* branches of the conditional with the program location of the conditional before the union operation. Thus any differences that are solely due to *readloc-writeloc* entries are removed and exponential growth is avoided. Given $\sigma = \{\theta_1, \ldots, \theta_k\}$ and a *block* which contains statements/control structures at program locations $pl = \{v_1, \ldots, v_i\}$, we define the operator $\clubsuit(\sigma, block, \mu) = \left\{ \theta_i|_{pl}^{\mu} \;\middle|\; \theta_i \in \sigma \right\}$, which performs the required replacements in the heap

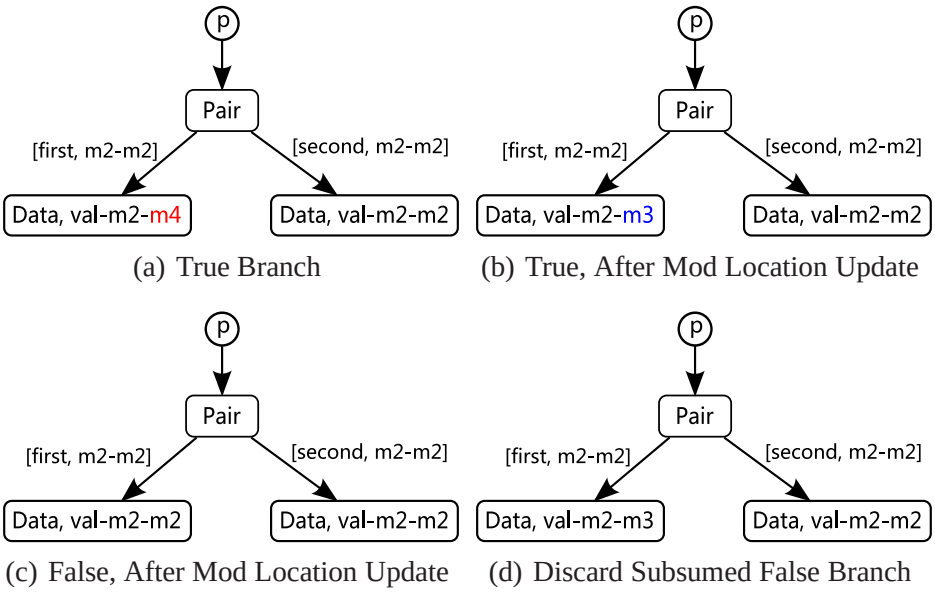
graph models. With this definition the improved semantics for the conditional operation (at program location κ) are:

$$\mathscr{S}[\![if(b)\ block_t\ else\ block_f]\!]\sigma =$$
$$\clubsuit(\mathscr{S}[\![block_t]\!](\sigma_{true}), block_t, \kappa) \cup \clubsuit(\mathscr{S}[\![block_f]\!](\sigma_{false}), block_f, \kappa)$$

Disjunctive Domain. To speed up program analysis we employ a *partially disjunctive domain* 7 which we use to discard elements in the abstract states (θ_i) that contain redundant read/write information. This is done by defining an order on the program locations based on their control-flow order. In general this order is not total (e.g. statement locations in the *true* and *false* branches of an `if` statement). However, our replacement of locations inside nested control-flow structures with the program location of the structure that contains them ensures that we can always compare the program locations that appear in the *readloc-writeloc* entries.

Analyze Conditional Example. Figure 11.3(a) is the abstract heap that approximates the state of the program after the *true* branch ($\mathscr{S}[\![block_t]\!](\sigma_{true})$), where the first element of the pair had the `val` field written. In the node that represents the `Data` object that was written we updated the *writeloc* entry to program location *m4* (where the write occurred, marked in red if color is available). Figure 11.3(b) shows the result of $\clubsuit(\mathscr{S}[\![block_t]\!](\sigma_{true}), block_t, m3)$, where we replaced the *readloc-writeloc* locations that appear in the *true* branch with the program location of the `if` statement (program location *m3*, shown in blue).

Figure 11.3(c) shows the abstract heap from the *false* branch where no write occurred ($\mathscr{S}[\![block_f]\!](\sigma_{false})$). The most recent *mod* location is unchanged (program location *m2*, where the object was initialized) in $\clubsuit(\mathscr{S}[\![block_f]\!](\sigma_{false}), block_f, m3)$ since program location *m2* is not nested in the conditional.



(a) True Branch (b) True, After Mod Location Update

(c) False, After Mod Location Update (d) Discard Subsumed False Branch

Figure 11.3: Updating Read/Write Locations At Control Flow Join

Given our order relation on the *use-mod* sites we can simplify the models resulting from the *true* and *false* branches into a single model shown in Figure 11.3(d). Intuitively the *may use-mod* information from the *true* branch indicates that the memory location at `p.first.val` *may* have been written at location *m3* (the `if` statement) or at some previous point in the program, while the result of the *false* branch indicates that the memory location at `p.first.val` *may* have been written at location *m2*. Since the possibility that the object may be written at or before program location *m2* is implied by the statement that the object *may* be written at or before program location *m3* we can safely discard the model from the *false* branch.

Abstract Loop Semantics. The semantics of a looping statement `while` at program location κ can be expressed in terms of accumulating all possible exit states. To do this we define the state of the heap at the loop test for the i^{th} iteration of the loop as:

$$\sigma_i = \begin{cases} \sigma & \text{if } i = 0 \\ \mathscr{S}[\![block]\!](\mathscr{S}[\![b]\!]_{true}(\sigma_{i-1})) & \text{otherwise} \end{cases}$$

Then we can define the semantics of the loop analysis as the union of all the possible exits from the loop with the read/write program locations that occur within the loop body replaced by the program location of the loop (κ). Formally:

$$\mathscr{S}[\![\mathit{while}(b)\ \mathit{block}]\!]\sigma = \bigcup \left\{ \clubsuit(\mathscr{S}[\![b]\!]_{\mathit{false}}(\sigma_i), \mathit{block}, \kappa) \;\middle|\; i \in \mathbb{N} \right\}$$

11.2.3 Read Write Locations in Interprocedural Analysis

In order to efficiently handle large programs we memoize the results of analyzing each method. At method call sites, if we were to naively compare the memoized heap models with the current call state the method specific *readloc-writeloc* entries we embed in the model would create many spurious inequalities. As an example consider the `swap` function from our running example. The `swap` method could be called from multiple locations in a program and at each of these call sites the `Pair` object may have a different *readloc-writeloc* entries for the `first` and `second` fields. If comparison is done in a naive manner these differences will result in spurious mismatches with memoized analysis values, forcing the method to be re-analyzed for each call.

To avoid this problem we anonymize the *readloc-writeloc* locations before attempting to find a match in the memo table. However, when doing this anonymization we need to ensure that we can figure out which locations in the result heap *may* have been read/written in the call and which *must* not have been read/written (and thus have the same *readloc-writeloc* entry as before the call).

Call Example. The anonymization and remapping operations are conceptually simple but without some intuition into how they function the definitions are difficult to follow. Thus, we first examine how the `swap` call is handled in the pair example. Figure 11.4 shows the steps that are taken to analyze the call at program location *m5* assuming that the memo table contains Subfigures 11.4(a) and 11.4(b) as a memoized result.

Figures 11.4(a) and 11.4(b) show that during the analysis of the `swap` method the analysis has determined the `first` and `second` fields have been read and written (the *readloc* and *writeloc* entries refer to program locations within the `swap` method, *s2*, *s3* and *s4*) but that the `val` fields are neither read nor written. The *readloc* and *writeloc* entries are the *modified-outside* value 0. Further, based on the identity tag sets we know that the object which was stored in the `first` field at the method entry (Figure 11.4(a)) and was given the identity tag 2 is stored in the `second` field at the method exit (Figure 11.4(b)). A similar situation holds for the object stored in the `second` field at the method entry, which was assigned the identity tag 3.

Figure 11.4(c) shows the state of the heap model at the call site (location *m5*) after we have added fresh tags (7, 8, and 9) to uniquely identify the nodes. After anonymizing the locations of the *readloc-writeloc* entries to the *modified-outside* value (0) we have the model shown in Figure 11.4(d), which is isomorphic (up to identity tags) to the model in our memo table, Figure 11.4(a).

During the anonymization we construct a map from the identity tags we added and the field identifiers to the *readloc-writeloc* entries in the caller scope that we are anonymizing. This gives us the map $ModM = \{(7, \texttt{first}) \rightarrow (m2, m2), (7, \texttt{second}) \rightarrow (m2, m2), (8, \texttt{val}) \rightarrow (m2, m3), (9, \texttt{val}) \rightarrow (m2, m2)\}$. Using the isomorphism from $\sigma_{in} \mapsto \sigma_{call}$ we have a map $\Pi = \{1 \rightarrow 7, 2 \rightarrow 8, 3 \rightarrow 9\}$.

Using these maps we transfer the read/write information from the call input to the memoized output, replacing any *readloc-writeloc* entries that refer to program locations in the callee body (`swap`) with the program location of the call site (program location *m5*) and replacing any occurrences of the *modified outside* value with the appropriate entry from *modM*. In Figure 11.4(b) the node with identify tag 2 has the *modified outside* value for the *readloc*/*writeloc* of the `val` field (`val-0-0`). To place the correct *readloc-writeloc* values into this node we look up the node that it maps to in the caller scope (via the Π map), which gives us the identity tag 8. Then we look up the caller scope *readloc-*

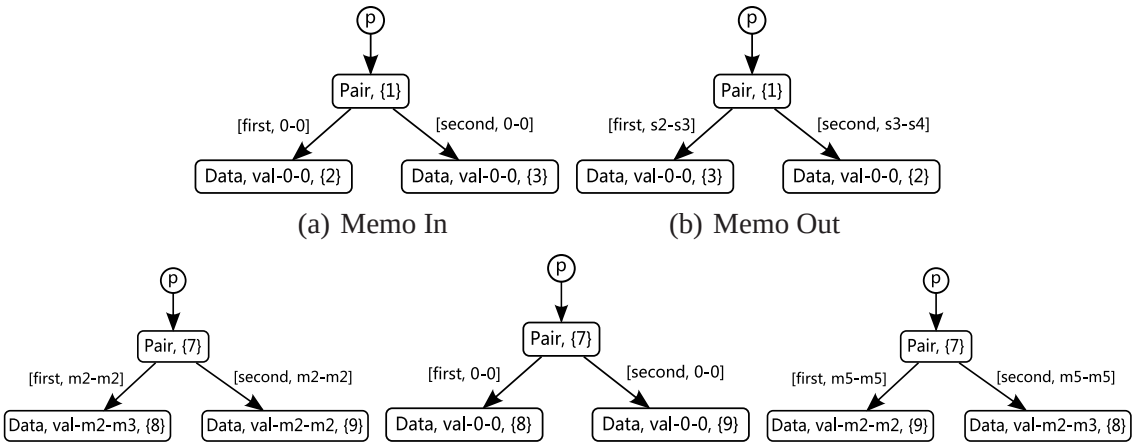


Figure 11.4: Mapping Through Memoization

writeloc information in the *modM* map, which gives us the read/write information for the field, `m2-m3`.

This remapping gives us the result in Figure 11.4(e), which shows that the object stored in the `second` field of the `Pair` object may have been written at program location *m3* but that the object stored in the `first` field has not been modified since initialization at program location *m2*. Thus, we can determine that the read from `p.first.val` is non-zero and the assertion will always succeed.

Dataflow Operations. For a method invocation at call site ℓ_{call} we give each node in the call state σ_{call} a unique tag $\kappa \in \mathbb{N}$, set the read/write location to the *modified outside* value and build a map $ModM : \mathbb{N} \times field \mapsto (\ell_r, \ell_w)$.

We then compare the anonymized version of σ_{call} with the entries in the memo table ignoring the read/write information. If a match $(\sigma_{in}, \sigma_{out})$ is found then there is a graph isomorphism $\Phi : \sigma_{in} \mapsto \sigma_{call}$. This isomorphism and the fact that the set of location tags in σ_{in} and σ_{out} are the same implicitly defines a map, $\Pi : \{\, \kappa \mid \kappa \textit{ a location tag } \in \sigma_{out}\} \mapsto \{\, \kappa' \mid \kappa' \textit{ a location tag } \in \sigma_{call}\}$. Using this map we can then compute the result of the call

by replacing any *readloc-writeloc* values (ℓ_x) for the fields in each node n with:

$$(\ell'_x) = \begin{cases} \ell_{call}, \text{ if } \ell'_x \text{ is a location in the callee method} \\ \max(\{n'.\ell_x \mid \kappa \in n.\text{identity} \wedge n' \in \sigma_{call} \wedge \Pi(\kappa) \in n'.\text{identity}\}), \text{ otherwise} \end{cases}$$

11.3 Case Studies with Data Dependence:

`Em3d`. The first application of the read/write dependence information we look at is performing thread-level parallelization of the `em3d` benchmark. In Figure 11.6 we show the code for updating the `value` field of a single `ENode` object. By applying our read/write analysis we obtain the model in Figure 11.5 at the end of the method body. We see that some object from the list of magnetic field nodes has had the `value` field both read and written in the loop, *readloc* = *c2* and *writeloc* = *c2* (marked in red if color is available), while there have been reads from the `coeffs` and `fromN` pointer fields, *readloc* = *c2* (marked in green), *writeloc* = 0. The pointers in the `fromN` array have also been read in order to access the `value` fields in the `ENode` objects in the opposite field, which have been read but not written (*readloc* = *c2*, *writeloc* = 0).

Using this information, the fact that each reference in the linked list (`LinkList`) of `ENode` objects refers to a unique object (the edge is *np*, the omitted default interference value) and the linear loop iteration, allows us to determine that each magnetic `ENode` object is written on a single iteration of the main update loop, program location *e2*, in Figure 11.7, which calls `computeNewValue`. Given this information it is valid to thread parallelize this loop (and to vectorize the loop in `computeNewValue`). Doing so results in a speedup of 3.21 on our quad-core test machine.

`BH`. Figure 11.8 shows the model that the analysis computes for the heap-based read/write information in the `hackGravity` method of the *Barnes-Hut* benchmark. For clarity we

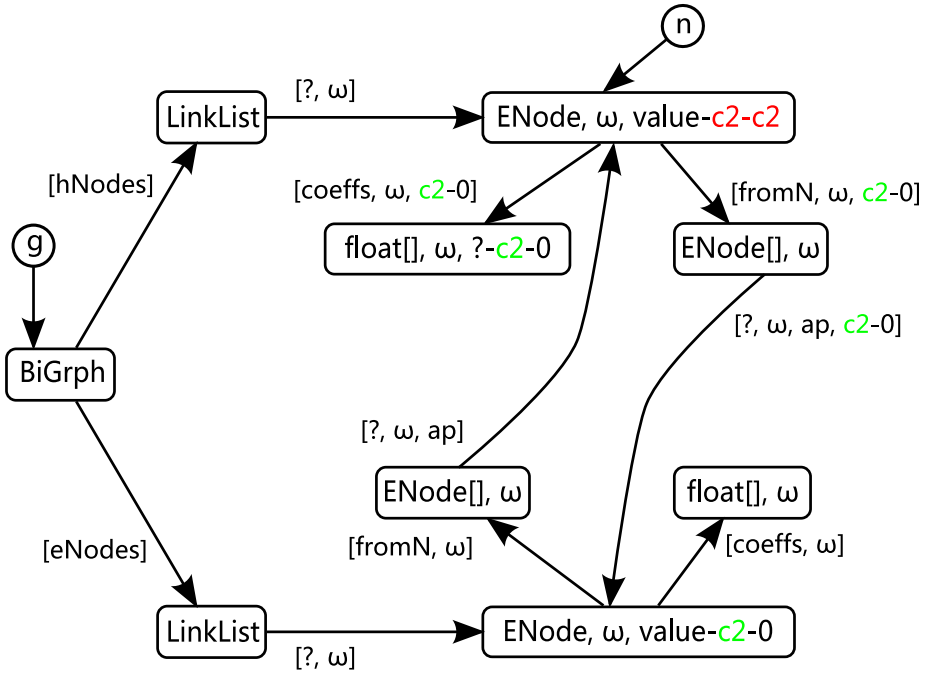


Figure 11.5: Em3d With Read/Write Info

```
c1  static void computeNewValue(ENode n) {
c2    for(int i = 0; i < n.fromCount; i++)
c3      n.value -= n.coeffs[i] * n.fromN[i].value;
c4  }
```

Figure 11.6: `Compute` (From em3d)

have simplified the heap structure in areas that are not relevant to this example.

The `bh` program performs a *fast-multipole* algorithm on the gravitational interaction between a set of bodies (the `Body` objects) and uses a space decomposition tree of `Cell` objects each of which has a `Vector` containing a subtree or a reference to the `Body` objects. The program also keeps two vectors for accessing the bodies, `bodyTab` and `bodyTabRev`. Figure 11.8 shows the state of the heap model after the loop body (Figure 11.9) that contains the majority of the computation in `bh`. This loop takes each `Body`

```
e1  for(int i = 0; i < this.hNodes.size(); ++i)
e2    computeNewValue((ENode) this.hNodes.get(i));
```

Figure 11.7: Main Em3d Compute Loop

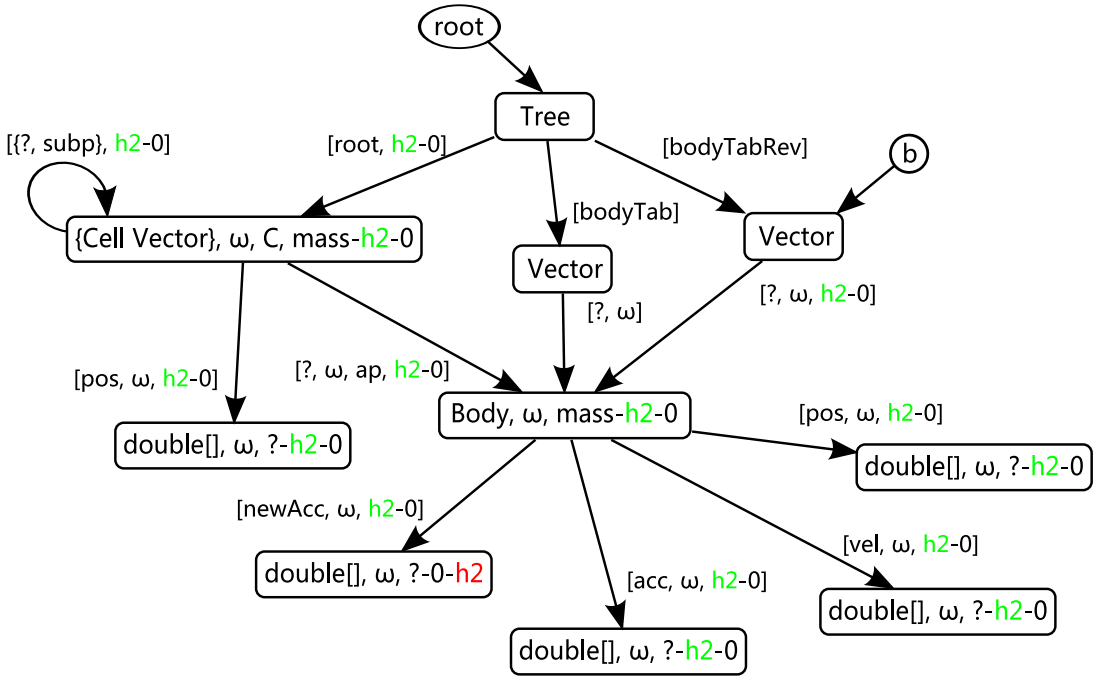


Figure 11.8: BH With Read/Write Info

object and walks the space decomposition tree (the `root` field) to determine a new acceleration value for the `Body` object (stored in the `newAcc` field).

Our analysis is not able to precisely resolve the construction of the space decomposition tree and conservatively assumes it may be a cyclic structure (shown by the `C` in the node representing the `Cell` objects). However, the analysis is able to determine that the `Cell` objects and the `Body` objects represent distinct regions in the program. This piece of information combined with the observation that the space decomposition tree is only read in the loop body (all the *readloc* entries set to *h2*, marked in green, and the *write-*

```
h1  Iterator b = this.bodyTabRev.iterator();
h2  while(b.hasNext())
h3    ((Body) b.next()).hackGravity(rsize, root);
```

Figure 11.9: Main Update, Gravity Computation

loc entries set to 0), that the only part of the heap which is modified is never read (the `double[]` stored in the `newAcc` field, *writeloc* = *h2*, set to red), and that the collection being indexed over (the `Vector` referred to by the `bodyTabRev` field) does not have multiple references to the same object (the *?* edge is *np*, the omitted default interference value), is sufficient to ensure that there are no heap-carried dependence in this loop. Thus, we can safely thread-parallelize the loop body, achieving a factor of 2.98 speedup on our test machine.

Chapter 12

Related Work

In this chapter we look at the most relevant previous work on the heap analysis problems that we are interested in. In particular we focus on the work on shape analysis and only briefly cover analysis techniques that are limited to computing points-to information. A comprehensive survey of the early work on pointer analysis can be found in [30]. We begin by briefly reviewing some of the work on *points-to* analysis as it is the most commonly used heap analysis technique despite the lack of precision in the results. We then examine the related work on more powerful shape analysis techniques. As all of the shape analysis approaches are fundamentally attempting to capture the same set of properties we categorize them based on how the properties of interest are represented: using a logical language with reachability predicates, model based approaches using graphs, or separation logic.

12.1 Points-to

Early work on analyzing the program heap focused on identifying *points-to* sets (set of variables that may point to the same location in memory). The most prominent of these papers are from Anderson [2] and Steensgaard [63]. These papers utilize either equiva-

lence or subset relations on the sets of program variables to compute a global points-to relation for the entire program.

While these methods are very efficient and produce acceptable results for some applications, the fact that they compute a single points-to relation for the entire program results in a very conservative approximation of the heap state. Some work has been done on computing similar points-to sets in a flow-sensitive manner [32] but experimental results indicate that the crude base domain makes the improvements from the more precise data flow minimal. In particular an experimental comparison of the results from Steensgaard's analysis and the results of flow-sensitive analysis demonstrates that there is little improvement in accuracy in points-to set metrics and no improvement in the results of the analysis clients (mod/ref, reaching definitions, and interprocedural constant propagation) [31].

The accuracy issues with these approaches arise from three areas. The first is obviously the use of a single program-wide points-to set, which forces a very conservative approximation. However, even when points-to sets are computed on a statement by statement basis, the results are still quite conservative. This is of course due to the imprecision in what the domain can represent (only points-to relations between variables) but also due to the inability of the domain to precisely simulate the effects of the program statements. For instance, in a linked list manipulation, statements of the form `x = y.next` often appear. If we understand that the heap object that `y` refers to is a *list* then it is clear that `x` and `y` cannot point to the same object after the assignment. However, the simple points-to set representation cannot express this connectivity information and so the analysis must conservatively assume (since there is no way to distinguish between a list and a cycle) that `x` and `y` may alias after the statement.

12.2 Logic Formula Based Approaches

12.2.1 Shape Predicate Analysis

A simple technique for using reachability predicates to model the shape and structure of the heap is presented in [20]. This work looked at using interference and connectivity via binary predicates over the set of variables in the program to model the *shape* of the portion of a heap reachable from each variable by using unary shape predicates. Given a variable *v* the unary shape predicates (*Tree(v)*, *DAG(v)*, and *Cycle(v)*) enabled the analysis to model how the program could traverse the sections of the heap reachable from each program variable. Binary relations on pairs of variables tracked the possibility that there was a path from the target of one variable to the target of the other variable (reachability), and if there was an object on the heap such that there was a path from both variables to this object (interference).

While this approach provides useful information for programs that build simple recursive structures, the connectivity predicates and shape properties are not sufficient to handle programs where regions would be temporarily transformed into more complex shapes (a DAG or cycle) before being returned to their original simpler shape. In particular when analyzing a program that performs a child swap in a binary tree, the tree is turned into a DAG during the intermediate steps before being restored to the tree shape. This approach is unable to capture this temporary transformation and cannot determine that the tree property is restored. The problem arises since the heap model has determined that there is a DAG-like structure but cannot determine what objects are involved in this DAG shape, thus the analysis cannot be certain that the DAG was broken after the swap.

The other major problem that arises with this approach is that the shapes are defined over very coarse sections of the heap (the portion of the heap reachable from a given variable). Thus, if a singleton design pattern is used, the analysis will determine that some

structure may have a DAG layout while a more sophisticated analysis could determine that there is a tree where the leaves may be shared singleton objects. The shape analysis in this thesis avoids these problems by giving each node in the heap graph a shape, which allows the analysis to model a tree of shared leaf objects precisely.

12.2.2 Path-Based Approaches

A variation of the shape-based approach is to track the connectivity properties of access paths in the heap instead of just tracking the connectivity of the variables. Tracking the connectivity of entire access paths avoids the need to track the shape of the heap reachable from each variable since the required connectivity information is explicitly tracked in the model.

A paper by Deutsch [18] introduced the idea of representing paths as regular expressions over the field identifiers in the program. As with Ghiya's approach, this work uses a binary relation to track the connectivity and interference relations between objects reachable from variables. However, instead of just tracking if these paths exist, Deutsch's approach also tracks the paths (via regular expressions over the fields in the program) that are taken from the source variables to the shared objects. This enables the analysis to track at what point in the heap the connections occur. However, like Ghiya's approach it cannot accurately model situations where the heap is temporarily transformed into a *dag* or *cycle* and then back into a *list* or *tree*.

In more recent work [25] Gulwani built on the ideas in [18]. In particular he utilized a much more powerful domain for modeling the access paths in the program. This allows the analysis to avoid many of the accuracy problems encountered by Deutsch. However, the approach still requires the modeling of several binary relations over the set of access paths in the heap (which can be a very large number) and a substantial number of tuples in the relation may need to be updated for each program operation. Thus, there are serious

questions about the performance of these approaches (and the preliminary results reported in the literature indicate that these techniques do indeed have substantial performance problems).

12.2.3 TVLA (Three-Valued Logic Analysis)

The Three-Valued Logic Analysis system (TVLA) [59, 60, 66] is based on a parametric framework for evaluating predicates in three-valued logic (truth values may be *true*, *false*, and *unknown*). This work provides an extensible parametric framework for adding additional predicates to the standard set of reachability predicates used in other work. This provides a much richer language for describing the heap. In addition to the construction of a parametric framework that can be extended via the introduction of new predicates as needed, the TVLA analysis system also introduced a number of novel and powerful concepts for analyzing the heap. The most important concepts are materialization and the focus operator, both of which we have translated in Chapters 8 and 6 into versions that are suitable for use in a graph-based model. These two ideas enable the shape analysis to transform summarized regions of the heap into a more explicit representation that can be modeled more accurately. This transformation enables the analysis to successfully model events like the tree child swap since the analysis can isolate the specific portion of the tree that temporarily violates the tree property and can thus identify the later operation that restores the tree property.

The TVLA model uses sets of unary and binary instrumentation predicates over program variables (and primed variables, which are introduced as needed to name specific locations in the heap) to track the interference and reachability of each variable pair. However, unlike previous analysis approaches, the TVLA analysis updates the set of locations that it tracks (the primed variables) to ensure that the targets of reads and writes can be explicitly identified and tracked. Thus, in the child swap example the TVLA analysis

generates a unique name for the node that is having its children swapped. By explicitly giving a name to this location the relation between it and its children can be explicitly and precisely tracked though all of the steps of the swap operation. This allows the analysis to identify that this object and one of its children temporarily create a *DAG* that is returned to a tree structure later in the swap operation. That is, since the locations of the objects involved in the *DAG* are explicitly identified and known to be unique, the TVLA analysis can safely infer that after the final write, the *tree* shape was restored since the analysis knows that the write must break the *DAG* and that this was the only *DAG* structure in the given section of the heap.

While this work introduced and explored a number of fundamental issues in modeling the program heap, the way that these properties was represented and the implementation of the analysis have severely limited the applicability of the results. From the standpoint of expressive power the particular set of predicates used in the TVLA work has limited the analysis to programs that only manipulate homogeneous recursive structures such as lists or trees. Thus, all of the programs discussed in the case studies section are beyond the capabilities of the TVLA analysis as presented in the published results. Further, the parametric construction of the analysis system provides significant flexibility to this analysis system, however this flexibility when coupled with the relational approach (the extensive use of binary relations) results in a computationally expensive analysis. Thus, the experimental results are limited to small benchmarks on the order of a few hundred lines of code and even then the analysis times are in the 10s–100s of seconds, making the analysis impractical for use in an optimizing compiler.

Significant Extensions. The TVLA research has also investigated important aspects of heap analysis techniques when dealing with function calls and container objects. We have again built on and adapted these concepts to work with our heap-based analysis, see Chapters 9 and 10.

The most prominent of the papers on interprocedural analysis focus on the concept of using cutpoints [55, 56], which split the heap into the portion that can be affected by the local call and the portion that cannot be modified, to perform more efficient interprocedural analysis. The concept of partitioning the abstract model, via a *project* operation, into a component that is visible in the callee method and a component that is out of the scope of the callee method is commonly used in abstract interpretation based approaches to improve the efficiency of the program analysis [5, 6, 46, 47, 51]. The ability to discard the irrelevant portions of the heap makes the use of memoization of analysis results significantly more effective and reduces the size of the model that the local analysis must deal with. These two features are of substantial importance to ensuring that the heap analysis is scalable. In particular, without the use of the improvements to the memoization results interprocedural heap analysis is intractable on even moderate-sized programs.

In [22] the idea of refining a summary representation into a more explicit representation was used to model scalar arrays. In this work the array is partitioned into multiple sections based on the integer indexing that the program is using to traverse the array. Each of these sections can then be modeled independently of the others. Thus, the analysis is capable of understanding programs that iteratively process arrays since it can maintain information separately for each segment of the array that is being processed (the portion that has been processed, the single element that is being modified, and the section that remains to be processed).

12.3 Model-Based Approaches with Graphs

A natural way to model how objects in the heap are connected is to use a graph. This has a natural relation to the structure of the concrete heap, which is often thought of as a graph. In particular this model allows the analysis to handle situations like the analysis of the `x = y.next` statement. In this case the graph based analysis (hopefully) will be able to

determine that the node representing the object `y` refers to and the node representing the object referred to by `y.next` are different and thus the analysis can conclude that `x` and `y` do not alias after the statement.

Although, intuitively, using a graph-based model seems like an ideal base for analyzing the heap, in practice there are a number of issues that prevented early work on using this model from being effective. The two most prominent papers on using a graph structure to model the heap are [9, 38], which we use as the basis for our model in Chapter 2. These papers outline the basic concepts of how a graph-based heap model works and provide examples of the issues that prevent their application in practice (issues that we have addressed in this thesis). Despite the introduction of some additional instrumentation to the graphs these papers, surprisingly, do not make the connection between these properties and the ability to parametrically extend the model with additional label properties as needed to address the problems (as we do in Chapter 3).

The most obvious problem is how these systems deal with recursive data structures and deciding which objects should be represented by each node in the graph. In particular [9] explained the difficulty in determining if a given node with a self edge(s) represents a list, a tree, or some cyclic structure. Although they presented a technique to differentiate between cyclic and acyclic structures, it was inadequate to deal with more general problems, such as differentiating lists from trees. This issue is resolved by the introduction of *shape* properties in Section 3.2, which allow us to precisely resolve all of these structures.

The difficulty in determining which objects should be grouped together is discussed in some detail in [9]. The first approach presented in the paper relies on using the allocation site to decide which node abstracts a newly allocated object. This results in all of the objects allocated at a given static site being represented by the same node even if dynamically they are in distinct data structures. For example if all `ListNode` objects are allocated from a single constructor method then all lists will be represented by the same node (even though there may be many distinct lists used in the program). Other ap-

proaches outlined in the paper create a new node for each allocation and then at join points use some heuristics to determine an *optimal* grouping of the nodes based on connectivity. The approach in this thesis, Chapter 6, presents a much more precise characterization of which nodes represent *logically related* sections of the heap, allowing the analysis to both identify and group similar objects together, while keeping the representations of unrelated data structures distinct in the model.

Even when the analysis in [9] was able to successfully identify a recursive data structure it was then unable to precisely model how this structure was updated during the execution of the program. In particular, if we remove an element from a list, the single-node representation of the list does not contain any information about the relative orders of the head of the list, the variable traversing the list, and the element to be removed. Thus, the destructive assignment removing the list element must be conservatively assumed to create a cycle. This particular problem was addressed in later work [59, 60] using logical formula based models and we adapted these solutions to the graph based model in Section 8.2.

While this early work on using graphs to model the heap had many issues that prevented its use in practice, it serves as an excellent basis to build on. In particular the research provides an excellent evaluation of the approach including analysis of the issues we described above, which greatly assisted in the construction of the model in this thesis.

12.4 Separation Logic

Separation logic was introduced by Reynolds, Ishtiaq, and O'Hearn [36, 54] as a variation on the logic of bunched implications [50]. The major advantage that separation logic has over using first order logic or simple predicates to model the program heap is the notion of the separating conjunction ($*$). This conjunction enables the logic to compositionally reason about the heap using frame rules which are driven by the separating conjunction.

The $*$ operator is used to denote that two regions of the heap are disjoint and thus the modifications to one of these regions cannot affect the other. This allows the logic to avoid the *frame problem*, thus significantly reducing the amount of work needed in simulating the effects of program statements on the heap model.

Recent work on using separation logic for analyzing the heap has looked at limited fragments of the full logic (to keep the analysis tractable), focusing on developing inductive predicates that can be used to model recursive structures [3, 24, 26, 68] and how these regions are assembled into composite structures [3, 68]. However, this work assumes that more complex sharing, such as an array that contains aliasing pointers, does not occur in the program. Thus, these techniques are limited in the range of programs that can be analyzed. Thus, again they cannot be used to analyze the majority of the programs that are included in the case studies from Chapter 5.

Thus, this work has a number of interesting applications where the restrictions placed on sharing and the types of data structures that are used are acceptable, in particular [3, 68] have focused on device driver verification. However, these types of restrictions are not acceptable when attempting to support a general purpose compilation system and attempting to use the full logic is too computationally expensive to be feasible. Thus, we see the work on separation logic and the analysis presented in this paper as complementary branches of work (both in optimization and verification applications) where the analysis in this thesis can be used as a forward pass to identify the major themes of the programs behavior, followed by an on-demand separation logic based approach to improve the precision where needed.

Chapter 13

Conclusion and Future Work

This thesis introduced a heap analysis technique that is designed to provide information on a range of properties (connectivity, aliasing, collection emptiness, data dependence, etc.) that are useful for program optimization and to do so in an efficient manner.

In order to accomplish this we began by introducing a parametric graph-based abstract heap model. Based on our experimentation with this model and the examination of the program properties we want to capture, we identified a number of properties, described in Chapter 3, that we used to augment the model. In Chapter 8 we showed how these properties can be used to ensure that the effects of the program operations on the state of the heap are simulated efficiently and precisely. Finally, in Chapter 6 we presented a normal form that allows us to identify recursive heap structures and to identify relations between the various data structures on the heap.

With these techniques providing a base framework for the heap analysis, we looked at how to efficiently perform interprocedural analysis in object-oriented programs (Chapter 9). In Chapter 10 we presented a novel technique for analyzing the collection libraries that are extensively used in modern Java programs. Then in Chapter 11 we showed how to extend the analysis to enable the explicit computation of heap-carried data dependence

information.

While each of these constructions is an interesting contribution on its own, the primary motivation for the development of each of them was to produce an analysis technique that is capable of precisely and efficiently analyzing real-world Java programs. To evaluate the suitability of the approach presented in this thesis for achieving this objective we devoted a substantial amount of effort to producing a Java front-end infrastructure and implementing the analysis itself. The result is an analysis system that can handle the majority of the Java 1.4 language and the most commonly used standard libraries in `java.lang`, `java.util`, and `java.io`.

In Chapter 5 we examined a number of case studies which demonstrate the precision and utility of the information produced by the analysis for optimizing a number of programs. In this chapter we present an empirical evaluation of the cost of analyzing these programs and look in more detail at using the information provided by the analysis to thread-level parallelize the benchmark programs. We then conclude with a look at the planned future work.

13.1 Evaluation

Benchmarks. To evaluate the efficiency and precision of the analysis we have collected a number of benchmarks that cover a broad range of heap structures and algorithms to ensure that the results reflect the utility of the analysis as a general purpose tool for optimizing Java programs. In particular we have selected several of the programs from the SPECjvm98 [62] suite, and the entire non-trivial JOlden [7] suite.

Our selections from SPECjvm98 are the benchmarks raytrace, a raytracer, modified to be single threaded, compress, a *Lempel-Ziv* text compression program, and db, an in memory database. These benchmarks are realistic applications from a standard Java benchmark

suite that build and use a number of interesting heap structures. Neither db nor raytrace have been successfully analyzed by any existing shape analysis techniques.

The JOlden suite contains pointer-intensive kernels (derived from the Olden benchmarks [8]) that make use of recursive procedures, inheritance, and virtual methods. We further modified the benchmarks taken from the JOlden suite to use modern Java programming idioms and addressed major concerns raised in the literature [69] about some of their deficiencies. The Olden benchmark suite was introduced in 1995 as a set of challenge problems to assess the effectiveness of parallelizing compilers and parallel architectures on programs that make extensive use of dynamically allocated data structures. In the intervening years a substantial number of papers have used benchmarks from this suite to evaluate various analysis and parallelization techniques. Despite the wide availability (and use of these benchmarks for evaluating compiler optimizations) a number of the more interesting programs (em3d, health, voronoi, and bh) have not been successfully analyzed by any other existing shape analysis techniques.

Results. The analysis algorithm was written in C++ and compiled using MSVC 8.0. The analysis as well as the parallelization benchmarks were run on a 2.6 GHz Intel quad-core machine (although the analysis is single-threaded) with 4 GB of RAM (although memory consumption never exceeded 160 MB). The benchmark code and the tool can be downloaded from [48].

The shape and sharing information from our analysis was used to identify loops and recursive calls that read from/write to disjoint sections of the heap (as described for several of the case studies in Chapter 5). This allowed us to parallelize the benchmarks to use multiple threads in loops and calls [21,29] to exploit the four cores of the test machine. The *Speedup* column in Table 13.1 shows the results. Some of the benchmarks (mst, compress, db) do not use any algorithms that can be parallelized using the simple techniques we are using for parallelization; we mark these with *NA*. In all but one of the remaining

benchmarks, we are able to achieve a significant performance improvement (up to 3.25× on power).

For each of the benchmarks we provide a brief description of some of the major structures/features that are in the program. We mention the major data structures used (Trees, Lists of Lists, Cycles, etc.) and if the program heavily modifies the data structures (w/ Mod). Some of the benchmarks have slightly more nuanced structures —mst and voronoi which build globally cyclic structures that have significant local structure, bh which has a complex space-decomposition tree and relations with the Body objects, and raytrace which builds a large multi-component structure which has cyclic structures, tree structures, and substantial sharing throughout. We also note that tsp, mst, and voronoi all begin with tree structures and process them building up a final cyclic structure during the program. Thus, these benchmarks exercise a wide range of features in the analysis based on the types of structures built, modification of these structures, sharing of the structures, use of multi-component structures, and the use of arrays/collections.[1]

To assess the accuracy of the analysis independently of the utility of the information for parallelization, we report, in the *Shape* column of Table 13.1, the results of the analysis technique. We use three categories for the accuracy of the analysis. Y(es) means the analysis was able to provide shape and sharing information for all of the relevant heap structures in the program. P(artial) means the analysis was able to determine the precise shape and sharing for some of the data structures but that some important properties were missed. N(o) means the analysis failed to precisely identify the shape/sharing information for a substantial portion of the heap data structures.

Our experiments demonstrate that the analysis method presented in this thesis can be used to efficiently and precisely analyze common programming idioms that build, share, and modify non-trivial data structures. While accurate, the techniques also scales to real

[1] Analysis results for the input/output states for a number of key methods in these benchmarks can be obtained at `http://www.cs.unm.edu/~marron/software/software.html`.

Benchmark	LOC	Description	Time	Shape	Speedup
bisort	560	Tree w/ Mod	0.26s	Y	2.38
mst	668	Cycle w/ Struct.	0.12s	Y	NA
tsp	910	Tree to Cycle	0.15s	Y	3.16
em3d	1103	Bipartite Graph	0.31s	Y	2.85
perimeter	1114	Cycle	0.91s	P	1.00
health	1269	Tree w/ Mod	1.25s	Y	3.21
voronoi	1324	Cycle w/ Struct.	1.80s	Y	2.43
power	1752	Lists of Lists	0.36s	Y	3.25
bh	2304	N-Body Sim. w/ Mod	1.84s	P	3.09
compress	1722	Arrays	0.29s	Y	NA
db	1985	Shared/Mod Arrays	1.42s	Y	NA
raytrace	5809	Shared/Cycle/Tree	37.09s	Y	2.74

Figure 13.1: LOC is the size of the program after transformation to MIL (including library stub code that must be analyzed), Shape reports if the heap information is correctly identified and Speedup reports if the Shape/Sharing information is useful, we report NA if the benchmark inherently has no useful thread-level parallelism.

programs, not just suitably chosen tricky program fragments.

Based on these results and the case studies in Chapter 5, the proposed approach provides a basis for an analysis technique that can be used in practice to provide detailed heap information for a range of optimization applications. In particular we have produced a (fairly robust) implementation of the analysis described in this thesis, which can support nearly the full Java language including many non-trivial features such as full support for the built-in collection classes, and verified that it can indeed produce the information that can be used in a range of optimization applications. Thus this work is a step in addressing the open problem of producing accurate and useful information about the shape and connectivity of the program heap and provides a solid foundation for continued work.

13.2 Future Work

While the work to date has produced a preliminary implementation of a heap analysis system that can handle the majority of the Java 1.4 language (and core libraries), our experimental results are currently limited to smaller programs (up to 6000 LOC). The primary cause of this limitation appears to be a combination of the immaturity of the analysis infrastructure (in particular the Java front end) and some of the engineering choices we made in the analysis implementation (in several cases we chose a simple implementation that does not scale well).

Thus, our first task in the future work is to improve the robustness of the analysis infrastructure so that we can perform a more complete evaluation of the analysis. This work focuses mainly on engineering efforts to improve the quality of the current implementation, removing inefficiencies in the analysis, more extensive testing, and improving the Java front end. Once completed we intend to implement a number of well-known optimization passes (common subexpression elimination, loop invariant code motion, method purity, and the identification of const-ness properties) to provide an empirical measure for the precision of the analysis (as opposed to the evaluation of the results by hand inspection).

The availability of a robust and scalable analysis tool and an automated system for evaluating the utility of the resulting information provides the basis needed to address the final objective of producing a practical heap analysis. In particular, this platform will allow us to identify (by allowing us to run the analysis on more and larger programs) remaining precision and scalability issues. If, as I believe, there are few if any major remaining fundamental issues with the precision or scalability (at least up to 20-30KLOC) the focus of the future work on the heap analysis will shift to evaluating a range of variations on the domain and analysis algorithm (such as improved support for array indexing or modeling the size of collections).

In this thesis we also mention modeling the values in storage locations and in Chapter 11 we provided an example of how the explicit store heap model allows properties of heap locations to be tracked. We intend to examine how similar techniques can be used to model object allocation sites/lifetimes and scalar values stored in memory. In particular, once we are able to track the allocation sites for the objects abstracted by a given node, we can, in conjunction with the identity information used to track read/write dependence information, precisely track the flow of objects from allocation until they become garbage. With this ability we plan to investigate the potential to use queries in *temporal logics* [15, 16, 61] to understand the behavior of the program.

One open problem that we have not looked at in this thesis and that remains a challenge is how to precisely model threads. While the analysis can be extended to handle multi-threading in a simple but safe manner it is not clear how much of an impact it will have on the precision of the results.

Thus, while we have not achieved our final goal of a producing a fully general analysis (as the analysis framework is still immature and there are open questions about scalability to large programs), we believe that this thesis represents a substantial advance in the state of the art for analyzing the heap and has successfully met (and in many cases exceeded) the major objectives we had for this work. We have succeeded in developing a heap analysis method that is able to precisely model a wide range of heap properties that are useful for program optimization. We can support nearly the full Java language including many non-trivial features, such as full support for the standard Java collection classes. We have produced a (fairly robust) full implementation of this tool and verified that it can indeed produce information (in a computationally tractable manner) that is useful in optimization applications. Thus, this work addresses (at least for smaller programs) the long-standing open problem of producing accurate and useful information about the shape and connectivity of the program heap and provides a solid foundation for continued work to scale this solution up to much larger programs.

Appendices

Appendix A

User's Guide For MTSA

This chapter contains a brief user guide for the MSTA analysis tool demonstration that can be obtained at [48]. We cover the installation and structure of the files contained in the demonstration in Section A.1. Section A.2 provides an overview of how to run the analysis demonstration and what can be done with it. Finally Section A.3 outlines how the stepwise analysis can be used to explore the information computed by and the working of the analysis. As the analysis is a work in progress we appreciate any comments and bug reports, and would be very interested in suggestions for other benchmarks to add to our suite (email: `marron@cs.unm.edu`).

A.1 Install and Included Files

The zip file "mtsa_demo.zip" contains all the files that are needed to run the analysis. Inside the `mtsa_demo` directory is the "analyze" program which is the analysis executable for linux and a directory, `data`, which contains the source files and directories that the output of the analysis will be put into.

In the data directory are the subdirectories `fms_graphs` which is where the analysis will place the labeled storage shape graphs, `ir_output` which contains the MIL files that the analysis works on, the `java_output` directory which is used for the executable versions of the MIL programs, and the directory `src` which contains the original source for the benchmarks.

The directory `fms_graphs` contains two subdirectories `dot_files` which is where the analysis places the ".dot" files before running *graphviz* on them. The resulting ".png" files are placed in the `jpegs` directory. The `jpegs` directory is where you will find the *interesting* heaps that are computed when the demo is run (the heaps at entry/exit to methods that are of particular interest in a given benchmark) as well as the *display* figures that are output by the user from the *step analysis*, Section A.3.

The `java_output` directory is where the demo writes the MIL files as Java programs which are executable and the call graph that is computed for the program. The call graph is rendered using *graphviz* where each node is labeled with a method name and there is an edge to any others methods that may be called. We color the graph with green nodes/edges if the call is to a builtin library method and blue nodes/edges if they are part of a strongly connected call component. Currently due to some simplifications in the library stub codes the targets of the builtin calls are not precise. In particular when the semantics of an overridden method are identical to the parent implementation we did not implement the method in our builtin library (the `add` operation for the `List` and `Vector` classes both result in calls to the method implemented in the `AbstractList` class). This issues does not affect the correctness of the shape analysis but makes it unsafe to currently use the call graph information for optimization and thus we plan to address this problem shortly.

Finally the `src` directory contains the Java 1.4 source files that are passed to our frontend compiler (not included in this demo) to produce the MIL files that the analysis processes. We divide them into three groups: the list/tree microbenchmarks in `microBench`, the modified Jolden benchmarks in `myOlden` and the two other benchmarks db/raytrace

in `otherBench`.

A.2 Running the Demo

The demo binary is statically linked to minimize any external dependencies. To render the graphs we make calls to the `dot` program from the *graphviz* suite and use `gthumb` as an external display tool to visualize them. If these programs are not on your path the analysis will function correctly but you will need to convert the ".dot" files placed in `data/fms_graphs/dot_files` and/or display the figures placed in `data/fms_graphs/jpegs` in some other way.

The demo is run by executing "analyze.exe" from the `mtsa_demo` directory. This brings up the first of the benchmarks, "TList", and we are given the option of analyzing this benchmark (`y`) to do so or (`n`) to skip it and proceed to the next benchmark. If you select to analyze this benchmark the code will be loaded and you will be presented with the command prompt for the *step analysis*, Section A.3 describes this in detail. Typing "run" (or r) instructs the analysis to run to completion (displaying the call stack of the analysis) and will then print some statistics on the runtime and a note that the approximation heaps written for the input/output of the methods are the result of joining all call states (thus potentially producing suprious imprecision that is not present in the analysis) and that the output set is being filtered to only include select heaps from interesting methods. You will also be prompted to see if you want to have the Java version of the MIL code and call graph written to `data/java_output` (`w` to write, `n` to skip). The demo will then prompt you for analyzing the next benchmark and will continue this way though the entire set of benchmarks. The benchmarks that can be analyzed are:

TList: a micro-benchmark for analyzing singly linked lists. It includes appending, both shallow and deep copies of the lists, insertion, removal and reversal.

Tree: a micro-benchmark for analyzing trees. It includes shallow and deep copies, as well as a recursive child swap.

BH: the *Barnes-Hut* n-body simulation as discussed in Chapter 5.

BiSort: a bitonic sort of a tree of values, heavily exercising the ability of the analysis to model destructive updates on recursive structures.

Em3d: an electro-magnetic simulation as discussed in Chapter 5 that requires the analysis to model non-recursive cyclic structures, Java collections and arrays.

Health: a health care simulation that involves the manipulation (insertion, removal and processing) of the data entries in a number of linked lists which are stored in a non-trivially recursive structure (there are objects of several types in the structure).

MST: computes a minimum spanning tree and requires the analysis to model hashtables as well as complex cyclic structures.

Power: a simulation of a power grid as discussed in Chapter 5.

TSP: an approximation of the traveling salesman shortest path as discussed in Chapter 5 which requires the analysis to precisely model a tree structure while also handling an unstructured cyclic structure efficiently.

Voronoi: computes the voronoi diagram for a given set of points as described in Chapter 5.

DB: an in memory Java database that performs extensive manipulation of objects stored in multiple arrays.

Raytrace: a raytracing program that uses a complex space decomposition tree along with a range of shape objects, material objects and light sources that are combined into a large data structure.

A.3 Step Analysis Controls

The step analysis interface allows us to examine the state of the analysis at any given program point and to see how the abstract state is impacted by a given statement. The analysis processes the statements in a given method in pseudo-program order (inner loops are completed before returning to the outer loop, both branches of a conditional are analyzed before processing any statements after the conditional, etc.). This makes understanding the analysis though the step interface very intuitive. At any time "help" [h] will bring up a list of all commands.

Before each command the analysis prints the current abstract call stack (which method it is currently analyzing and which methods are pending on the result of analyzing this method) and the current statement that is about to be analyzed (this may be an atomic statement or an entire control flow structure such as a conditional). If we launch the analyzer and opt to analyze the "TList" program the analyzer tells us we are in the `main` routine at line 280. To show more of the method type "print" [p] which prints the entire body of the method (in the equivalent Java source). We can set breakpoints for the analysis using the "break" command. If we type "b 288" at the prompt we set a break point at the first call to the `makeListSize` method. To then run to this point we use the "run" [r] command.

While running to the breakpoint the analysis first executes the static initialization routines (indicated by displaying the names of the methods as it analyzes them and indenting the calls according to their depth in the abstract call stack). When we hit the breakpoint control is returned to the step analysis and we can step into the `makeListSize` method using the "stepinto" [si] command. The analyzer responds by printing the call into the method and then halts at the first statement of the called method body.

We can then proceed to the interior of the loop that constructs our linked list by setting the breakpoint at line 224, "b 224" and then running to that point "r". Once there

we can display the current state of the abstract heap by using the "display" [d] command. This opens a new viewer window (in gthumb) displaying the current state of the heap (which is not too interesting right now). The current line is another method call to `makeRandomDataNode` which we would rather not step into so we use the regular "step" [s] command which runs the analysis to the next statement regardless of calls (a number of calls will be shown). If we use the "step" command a few more times until we are at line 229 we can then use the "display" command to show the state of the abstract heap after the first iteration of the loop. The current state has the start of a linked list consisting of a single list object pointing to a single data object. If we continue stepping though line 230 the analysis will return to the first statement in the loop body (line 223). We can continue stepping though the loop body watching the analysis model the construction of the list until after a few iterations it has a model with a node representing the head of the list and a node with a list layout which represents the tail of the list.

After one more iteration the analysis will conclude that for the given state at entry to the loop it has covered all possible states that can occur in the loop body and will proceed to the statement after the loop (line 233). If we display the abstract state here we will see a single state of a single head node and a tail with a *List* layout which is a safe approximation of all possible lists built in the loop. Using the "step" command again will bring us to back to the main method.

At this point we can set a breakpoint at line 251 (inside the append method). This particular location allows us to demonstrate the power domain that is used. After setting the breakpoint "b 251" and running to the line "r" we can print the method body "p" and display the abstract state of the heap "d". In this case we have 3 possible states depending on the relations between the heads of the list `l` and the temp variable, `y` which depend on the length of `l`. The analysis uses 3 states to track the possibilities and thus there are 3 figures (`display0.png`, `display1.png` and `display2.png`) displayed to cover the cases.

At this point we leave you to continue exploring the program by returning to the main method ("b 301" followed by "r") We encourage you to explore some of the other methods in the `TList` micro-benchmark. Of particular interest are the list shallow/deep copies (lines 307, 308), the insertion/deletion operations (line 344, 360) and list reverse (line 376).

References

[1] J. Alves-Foss and F. S. Lam. Dynamic denotational semantics of java. In *Formal Syntax and Semantics of Java*, 1999.

[2] L. Anderson. Program analysis and specialization for the c programming language. Tech. report, PhD. Thesis, DIKU, report 94/19, 1994.

[3] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *Computer Aided Verification*, pages 178–192, 2007.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *Foundations of Software Technology and Theoretical Computer Science*, pages 97–109.

[5] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10:91–124, 1991.

[6] F. Bueno, M. G. de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. J. Stuckey. A model for inter-module analysis and optimizing compilation. pages 86–102, 2001.

[7] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, 2001.

[8] M. C. Carlisle and A. Rogers. Software caching and computation migration in olden. *SIGPLAN Notices*, 30(8):29–38, 1995.

[9] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[10] B.-C. Cheng and W. mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. pages 57–69, 2000.

[11] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *Symposium on Memory Management*, pages 85–96, 2004.

[12] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. In *Symposium on Memory Management*, pages 138–149, 2006.

[13] W.-N. Chin, F. Craciun, S. Qin, and M. C. Rinard. Region inference for an object-oriented language. In *Conference on Programming Language Design and Implementation*, pages 243–254, 2004.

[14] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[16] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[17] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[18] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In *Conference on Programming Language Design and Implementation*, pages 230–241, 1994.

[19] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Conference on Compiler Construction*, pages 82–93, 2000.

[20] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[21] R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in C programs with recursive data structures. In *Conference on Compiler Construction*, pages 159–173, 1998.

[22] D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *Symposium on Principles of Programming Languages*, pages 338–350, 2005.

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha. The java language specification second edition. http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html, 2000.

[24] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symposium*, pages 240–260, 2006.

[25] S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *Computer Aided Verification*, pages 379–392, 2007.

[26] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *Conference on Programming Language Design and Implementation*, pages 256–265, 2007.

[27] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: a static analysis for automatic individual object reclamation. In *Conference on Programming Language Design and Implementation*, pages 364–375, 2006.

[28] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Symposium on Principles of Programming Languages*, pages 310–323, 2005.

[29] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, 1990.

[30] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[31] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[32] M. Hind and A. Pioli. Which pointer analysis should I use? In *Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[33] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *Conference on Object-Oriented Programing, Aystems, Languages, and Applications*, pages 359–373, 2003.

[34] S. Horwitz, P. Pfeiffer, and T. W. Reps. Dependence analysis for pointer variables. In *Conference on Programming Language Design and Implementation*, pages 28–40, 1989.

[35] J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Conference on Programming Language Design and Implementation*, pages 218–229, 1994.

[36] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[37] Jolden Suite. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.

[38] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Symposium on Principles of Programming Languages*, pages 1–28, 1979.

[39] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Conference on Programming Language Design and Implementation*, pages 278–289, 1993.

[40] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. volume 27, pages 235–248, 1992.

[41] C. Lattner and V. Adve. Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis. Technical Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[42] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. pages 129–142, 2005.

[43] T. Lev-Ami, N. Immerman, and S. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *Computer Aided Verification*, pages 547–561, 2006.

[44] M. Méndez, J. Navas, and M. Hermenegildo. An efficient, parametric fixpoint algorithm for analysis of Java bytecode. In *BYTECODE*, pages 51–66, 2007.

[45] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[46] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Symposium on Principles of Programming Languages*, pages 330–341, 2004.

[47] K. Muthukumar and M. V. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2-3):315–347, 1992.

[48] MTSA Demo, August 2008. `http://www.cs.unm.edu/~marron`.

[49] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[50] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[51] G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. G. de la Banda, K. Marriott, and P. J. Stuckey. A generic framework for context-sensitive analysis of modular programs. In *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, pages 233–260. 2004.

[52] F. Qian and L. J. Hendren. An adaptive, region-based allocator for Java. In *Symposium on Memory Management*, pages 127–138, 2002.

[53] J. Qian, B. Xu, and H. Min. Interstatement must aliases for data dependence analysis of heap locations. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 17–24, 2007.

[54] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[55] N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Symposium on Principles of Programming Languages*, pages 296–309, 2005.

[56] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural functional shape analysis using local heaps. Technical Report 26, Tel Aviv University, Nov. 2004. Available at `http://www.math.tau.ac.il/~maon`.

[57] N. Rinetzky and S. Sagiv. Interprocedural shape analysis for recursive programs. In *Conference on Compiler Construction*, pages 133–149, 2001.

[58] R. Rugina and M. C. Rinard. Automatic parallelization of divide and conquer algorithms. In *Symposium on Principles and Practice of Parallel Programming*, pages 72–83, 1999.

[59] S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Symposium on Principles of Programming Languages*, pages 16–31, 1996.

[60] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[61] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Symposium on Principles of Programming Languages*, pages 38–48, 1998.

[62] Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. http://www.spec.org/jvm98.

[63] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[64] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programing Languages and Systems*, 13(2):181–210, 1991.

[65] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-oriented Programing, Systems, Languages, and Applications*, pages 187–206, 1999.

[66] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *Conference on Compiler Construction*, pages 1–17, 2000.

[67] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[68] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable shape analysis for systems code. In *Computer Aided Verification*, pages 385–398, 2008.

[69] C. Zilles. Benchmark health considered harmful. In *Computer Arch. News*, 2001.